SOLUTION TO THE GRIPPER ENVELOPE PROBLEM USING A PLANAR SWEEP

NAGW-1333

by

Henry L. Welch

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering Department
Troy, New York 12180-3590

April 1992

CIRSSE REPORT #111

Solution to the Gripper Envelope Problem Using a Planar Sweep

HENRY L. WELCH

Abstract -- The determination of sweep shadows is important when analyzing the potential interference effects of obstacles in a robotic environment and is well suited for application to the gripper envelope problem. This report presents techniques for generating the planar sweep shadow of polyhedral objects. Algorithmic details for computing the sweep shadow for both straight-line and simple-rotation trajectories are described. Methods for analyzing the resultant sweep shadow are also presented and the time complexity of the algorithms is discussed. Various test cases are provided in an appendix showing the actual results generated by software solving the problem.

Table of Contents

1.0 Introduction

The order in which an assembly is performed can drastically affect the overall time it takes to perform that assembly. A good ordering can reduce the number of assembly errors and manufacturing difficulties. For example, some alternate assembly sequences may require less fixturing or fewer changes of tools and grippers than others. It may be possible, by assembling different parts at different times, to develop sequences which have mating trajectories with fewer and more distant obstacles to avoid. Such a choice could thereby result in simpler and more reliable assembly operations.

One the goals of this report to answer some of the questions posed by the calculation of geometric feasibility. Geometric feasibility determines whether or not two subassemblies can be properly mated along a collision free trajectory. Related to this calculation is the determination of the gripper envelope, or the volume available for the gripper, during an assembly operation. The goal of these types of calculations is to determine the potential interference effects of obstacles in the environment.

The appendices serve as an aid for bridging the gap between the algorithmic details presented and the software package which implements these ideas.

2.0 Background

A significant body of research has been compiled in recent years which addresses the question of assembly sequence planning. In most cases the basic approach is to use the geometric relationships between parts and the idea of geometric feasibility to generate a list of all the possible feasible assembly sequences. Differences between the various approaches involve the method by which assembly sequences are represented and the degree and type of operator interaction required by the algorithm.

Two of the earliest attempts, by Ko and Lee [Ko] and Fox and Kempf [Fox], use a precedence graph to represent the relationship between the various assembly tasks and require the operator to supply all the geometric feasibility information. Further work by De Fazio and Whitney [De Fazio 87, De Fazio 88] uses directed graphs of assembly states and provides a consistent set of operator questions for determining geometric feasibility. Later, Homem de Mello and Sanderson [Homem 86, Homem 88] propose the use of AND/OR graphs of subassemblies to represent the assembly sequences and provide an algorithm for analyzing geometric feasibility.

A common element missing from all these approaches is the ability to rank the various assembly sequences so as to be able to determine the sequence most likely to be successful. Among the factors which may be used to rank sequences and specific operations in the sequence are the size and shape of the gripper envelope and the subassembly stability (in the presence of friction and gravity).

Figure 1 shows a proposed blueprint for the architecture of an assembly sequence planner. Many of the modules depicted exist, in one form or another, as research efforts throughout the literature [Welch]. The notable exception is the module for performing gripper envelope analysis. In this paper the issues related to determining the bounds on the gripper envelope are addressed.

2.1 The Gripper Envelope

Humans and robots are both similar in that each need an envelope or volume in which to perform actions. In most cases the volume required for an action is centered about the object being acted upon and changes as the operator moves the object past obstacles in the environment. There are

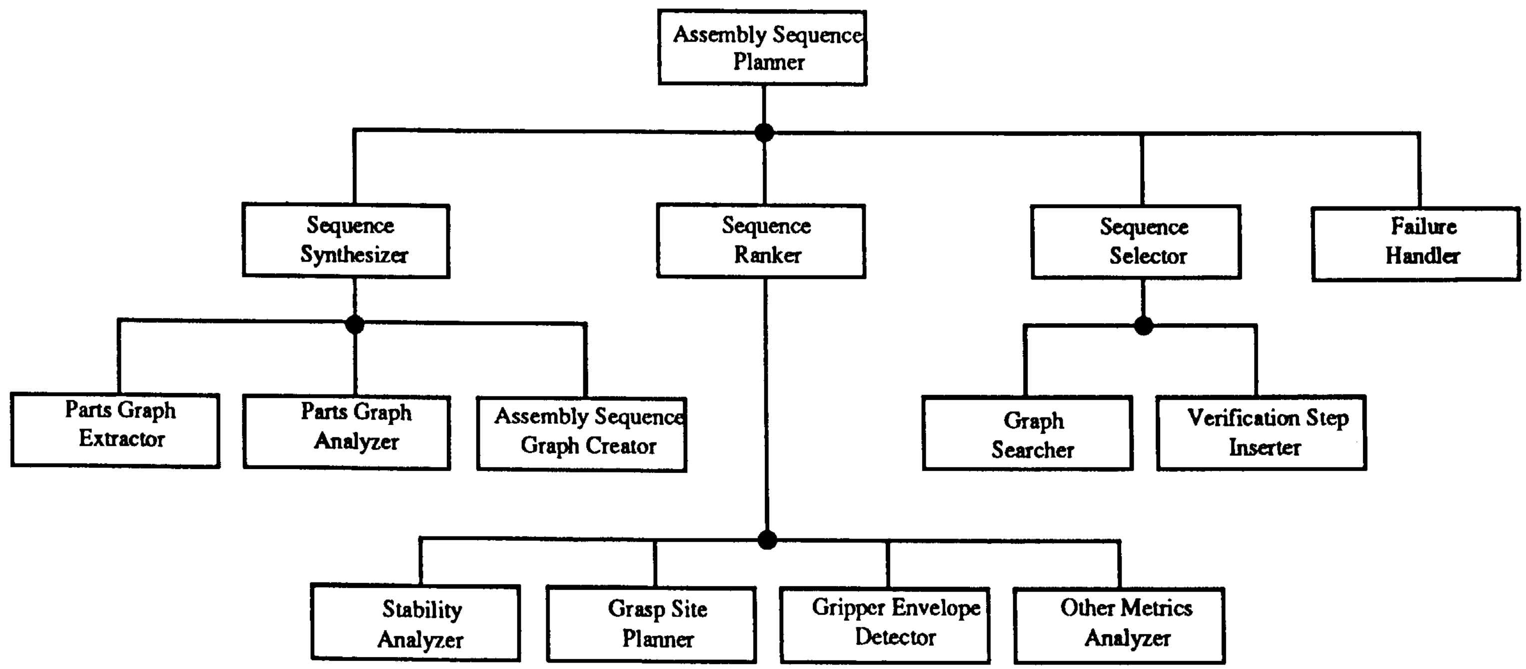


Figure 1 - Basic structure chart for an assembly sequence planner

two techniques which can be used to deal with the problem. The first requires full geometric data on the robot or agent performing the assembly. An algorithm for detecting collisions, for example, is one proposed by Mirolo and Pagello [Mirolo]. The second technique performs the motion without the presence of the agent and generates the volume not occupied by obstacles for later analysis. This second technique is the one being considered here because it provides a non-robot-specific solution.

Initially the calculation of the gripper envelope closely resembles the volume sweeping problem, as described by Wang and Wang [Wang], in which the obstacles in the environment sweep out volumes which cannot be occupied by either the part being mated or the mechanism moving the part. However, by limiting the types of motions that an object may follow, the problem can be simplified.

2.2 The Plane Sweep Algorithm

As is commonly the case in both robotics and other fields a problem is first partially solved by making restrictions to the shapes of geometric entities. In the field of robotics this usually involves restrictions to the types of paths possible in mating trajectories and to the shapes of objects being represented. Another common assumption in robotics is that the geometric relationship between that object and the hand/fingers of the agent remains fixed from the time an object is grasped until the time it is released. This eliminates many types of uncertainty that are difficult to model.

The most common limitation on the shapes of geometric entities is to limit objects to the set of convex polyhedrons as demonstrated by Mirolo and Pagello [Mirolo]. A less restrictive class of objects would be those possessing planar faces. This is a common technique in surface modeling and yields accurate approximations of most objects [see Blinn, Foley, and Turner]. This is the class of objects used throughout the rest of this paper.

From basic mechanics, it is known that all forms of motion can be broken up into two distinct components. The first component is tangential to the direction of motion and the second component is normal to it. Generalizing the concept to volume sweeping divides the volume being swept into a cross-sectional area perpendicular to the direction of sweeping and a distance along the direction of sweeping.

The cross-sectional area of the gripper with respect to the mating trajectory is constant due to the fixed relationship between the object and the robot agent's hand and the limitation of the mating trajectories to paths with uniform tangential components, . Examples of trajectory paths which fit this criterion are straight line segments, simple rotations, helical paths, and constant radius curves. Thus, by looking at the projected sweep shadow cast by obstacles in the direction of the path tangent, an estimate of the cross-section available for a robot's hand can be made.

3.0 Generation of Sweep Shadows

These shadows can be generated by sweeping a plane along the direction of the motion and marking the cross-sectional area of the obstacles encountered by the plane. Hence, the name "The Plane Sweep Algorithm." The next few sections describe the details for computing the projected shadows in the sweep plane.

3.1 Straight Line Trajectories

The generation of sweep shadows for straight line segments is straightforward. As described previously, the plane sweep algorithm generates these shadows by sweeping a plane along the direction of motion and determines which of the objects in the environment intersect with the plane and they do so. This problem is similar to projection in a cartesian coordinate system as shown in

Figure 2.a. In order to generate the proper shadow, three main functions need to be performed. These are the projection of data, the clipping of unwanted data from the projection, and the closure of faces and objects brought about by the clipping process. These functions are described below.

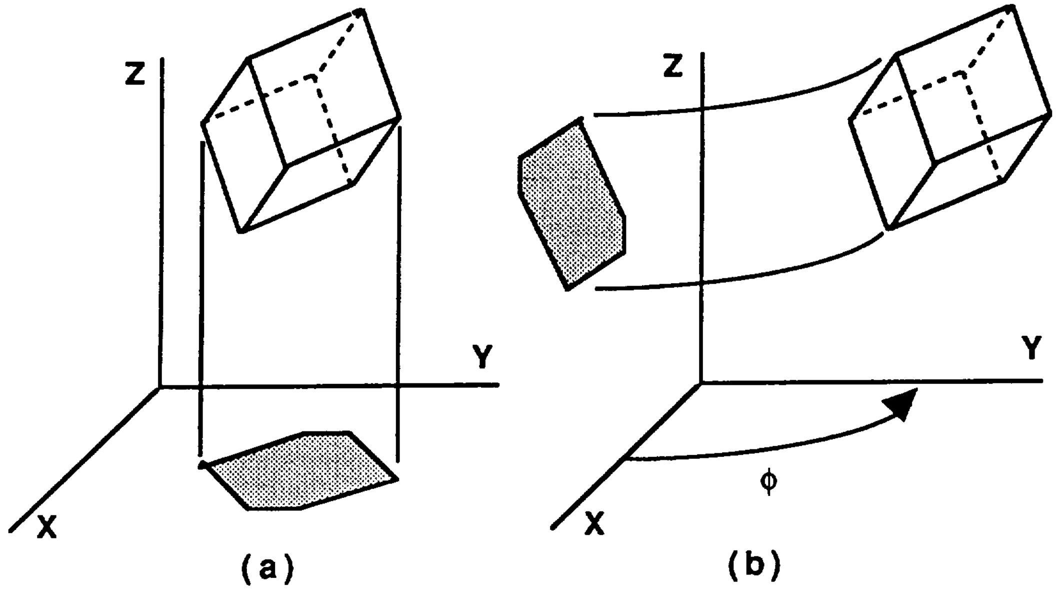


Figure 2. (a) Sweeping a plane along a straight line segment. Sweeping the XY-plane along the Z-axis can be accomplished by projecting the cube onto the XY-plane.
(b) Sweeping plane through a simple rotation. Rotating the XZ-plane about the Z-axis can be accomplished by projecting the cube along the ϕ direction.

3.1.1 Projection in Cartesian Coordinates

Projection in a cartesian coordinate system is the simplest form of data projection, especially if it is along one of the primary axes of that coordinate system. This is done by simply eliminating the coordinate of the axis that is in the direction of the projection. For simplicity, define the x-axis of the projection coordinate system as the direction of the motion and the origin of the projection coordinate system as the starting point of the motion. By defining other suitable axes for the y and z directions of the projection coordinate system the object data, as represented by P, can be converted to the projection coordinate system by using a coordinate transformation, T, as shown below [Selby, p. 369].

$$[P - P_r] = [a\ b\ c][T] \rightarrow [a\ b\ c] = [P - P_r]\,[T]^{-1}$$

3.1.2 Clipping Edges of Objects

Once the object data is converted to the new coordinate system it can be projected to the sweep plane by eliminating the x coordinate. This is not sufficient, though, to generate the correct sweep shadow since the sweep plane only traverses a finite segment of the x-axis. The removal of unwanted data is called clipping. The most common form of clipping performed in programming today is with respect to screen or window boundaries in computer graphics [Hearn, pp. 128-134]. The general approach is to clip the edges of an object with respect to one boundary at a time. In the case of the motion described above, the clipping planes are at $x = 0$ and $x = length$ (of the motion). This can be done by parameterizing each edge and removing the unwanted portions.

3.1.3 Closing Object Faces

There is a side effect to the clipping operation. This is caused by the portions of objects that are removed at the boundaries of the clipping region. Unless extra edges are added to the clipped object then the clipped object is left open at the boundary of the clipping region. This is corrected by the process of closure. The simplest way to close the face of an object is to traverse its edges in an orderly fashion. When an edge leaves the clipping region (i.e. outside of the area to be swept) this location is recorded. When the next edge is found which reenters the clipping region then a new edge is added which connects the point where the face left the clipping region and where it reenters. This automatically closes each face and object [Hearn].

3.2 Simple Rotations

The goal of the rotational sweep shadow problem is to compute the areas in the radius-height plane which are swept by a full plane sweeping through an angle in the specified rotational reference frame given the arbitrary rotational reference frame, the angle to rotate through, and the environment of obstacles. More specifically, given an arbitrary axis of rotation, center of the rotation space, a reference point defining the starting location of the sweep plane, and an angle to rotate the plane through, sweep the plane through the rotation and record all the areas on the plane that are *swept out* by obstacles in the environment.

The rotation of a full plane is necessary since it is unknown which half of the sweep space is important relative to the reference point. For example, consider a hand drill with a T shaped handle. While it is sufficient to represent its orientation by specifying the location of one side of the handle, both sides of the handle may encounter obstacles when it rotates.

Upon further consideration, this problem reduces to the projection of data (through the angular coordinate) in an arbitrary cylindrical coordinate system, as shown in Figure 2.b, with the added consideration of clipping at the initial and final location of the sweep plane. There are also some other complications brought about since the plane being swept is a full plane and not the half-plane usually associated with a cylindrical coordinate system. The two most obvious complications are the inclusion of data points containing both positive and negative radii (something not allowed in a strict cylindrical coordinate system) and, when the rotational angle is greater than π, points on an obstacle may appear in both the positive and negative sides of the sweep plane. These are some of the features which make this problem interesting.

3.2.1 The Basic Solution

Having defined the rotational sweep shadow problem, it is now possible to explore solutions to the problem. Since the sweeping operation reduces to projection through the angle of rotation in a cylindrical coordinate system, it is advantageous to use cylindrical coordinates in the solution of the problem. Since clipping and closure are also considerations, algorithms to accomplish these operations are needed. Figure 3 shows a simplified description of the algorithm to be used.

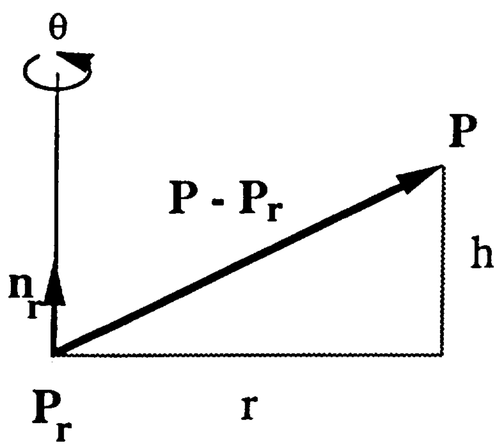
The next few sections describe in detail the various calculations necessary to implement the algorithm.

3.2.1.1 Defining the Cylindrical Coordinate System

The initial step in solving the rotational sweep shadow problem is to develop a representational formalism that simply and compactly defines the relevant data. Figure 4 shows how an arbitrary point in the environment can be represented in terms of a cylindrical coordinate system defined by the rotation.

```
procedure GENERATE_ROTATIONAL_SWEEP_SHADOW;
    specify and define the cylindrical coordinate system for the sweep;
    for each object in the environment;
        for each face on the object;
            for each edge on the face;
                convert the edge to the sweep coordinate system;
                sweep the edge and clip as necessary;
                close with the previous edge if necessary;
            close the face between the initial and final vertices;
end-procedure;
```

Figure 3. A simplified algorithm for solving the rotational sweep shadow problem.



$$h = (\mathbf{P} - \mathbf{P}_r) \cdot \mathbf{n}_r$$

$$r = \sqrt{(|\mathbf{P} - \mathbf{P}_r|^2 - h)^2}$$

Figure 4. An arbitrary cylindrical coordinate system defined by a spatial rotation.

One problem with the formalism of Figure 4 is that it does not specify the angular component of the point P. To do this, a cartesian coordinate system must be built around the cylindrical system so that a reference direction for the angular component can be defined. The first obvious choice of axis is to use the axis of rotation as the new cartesian z-axis. Specifying the new cartesian x- and y-axes requires more thought. Since a reference point representing the starting location of the rotation plane has already been specified it would be advantageous to use this to define the new cartesian x-axis. In general the direction from the center of the rotation to the reference point is not orthogonal to the already chosen z-axis. The solution to this is to use the Gram-Schmidt orthonormalization technique to specify the x-axis orthogonal to the existing z-axis [Hoffman, p. 280]. The y-axis then follows naturally in the right hand sense as the vector cross product between the z-axis and the x-axis.

3.2.1.2 Converting Data to the Sweep Reference System

Now that the sweep reference system is defined it is necessary to convert all the environment data to that coordinate system. Letting T represent a coordinate transformation matrix from the global coordinate system to the sweep reference system using the x-, y-, and z-axes defined in the previous section. A matrix notation can be used to convert the coordinate P, as defined in Figure 4, to the r, θ, and h of the sweep reference system by first finding the scalar components, $[a\ b\ c]$, of the position vector P in the sweep reference system as shown below [Selby, p. 369].

$$[P - P_r] = [a\ b\ c][T] \rightarrow [a\ b\ c] = [P - P_r]\,[T]^{-1}$$

$$r = \sqrt{a^2 + b^2}$$

$$\theta = tan^{-1}(\frac{b}{a})$$

$$h = c$$

3.2.1.3 Clipping Edges Against the Rotational Sweep Wedge

In a rectangular system the clipping of edges against a fixed planar boundary is straightforward and yields relatively few difficulties [Hearn, pp. 128-134]. Extending the idea of clipping to two parallel planar faces involves clipping against each of the planar faces individually. It would appear that the same basic idea can be used to clip an edge against the two constant θ-planes which bound the rotational sweep wedge; however, this is not entirely true.

If the calculation of the rotational sweep shadow involved only the rotation of a half-plane then clipping against the two constant θ sides of the wedge would sufficiently clip the end or ends of each edge. However, a consideration of both positive and negative wedges reveals that edges swept by the negative radius half of the sweep plane must be clipped to the opposite sides of the constant θ wedge boundaries. This requires clipping of each edge against four boundaries to solve the problem. It is also possible to clip each edge against the two positive radius wedge boundaries by rotating the edge data through π radians and then reclipping against the same two boundaries. This second technique is the one that is used here and is essentially the same as performing the clipping operations for two half plane rotational sweeps.

Another approach to the clipping problem involves clipping against both constant θ wedge boundaries simultaneously. This is not as straightforward as the rectangular case because of the wrap-around nature of angular data. (Angles greater than twice π are not possible.) In the rectangular case, an indication would be given as to whether an edge's endpoints are beyond one of the two boundaries and which of the two boundaries it is beyond. In the case of cylindrical data, it is not readily apparent whether the endpoint is beyond the $\theta = 0$ or $\theta = \theta_{max}$ wedge boundary. This effect can be counteracted by also considering the midpoint of each edge. Figure 5 shows all the possible arrangements of endpoints and midpoints for the case when the first endpoint as represented by $p1$ has a θ value greater than the second endpoint which is represented by $p3$ Solution of the problem when the role of the endpoints is reversed follows by symmetry.

If either of these techniques is used, a method is still needed to calculate the intersection point with the wedge boundary. By parameterizing the edge with endpoints p_1 and p_2 the following vector formulation is obtained:

$$[p_1 + m(p_2 - p_1)] = [a\ b\ c];\ 0 \leq m \leq 1$$

For the case of finding the intersection point with the $\theta = 0$ wedge boundary the coordinate b becomes zero. By introducing the vector d= [0 1 0] and by taking the vector dot product of d with both sides of the above equation the following condition results [Selby, p. 540]:

$$p_1 \cdot d + m(p_2 \cdot d - p_1 \cdot d) = 0$$

Solving for m yields:

$$m = \frac{p_1 \cdot d}{p_1 \cdot d - p_2 \cdot d}$$

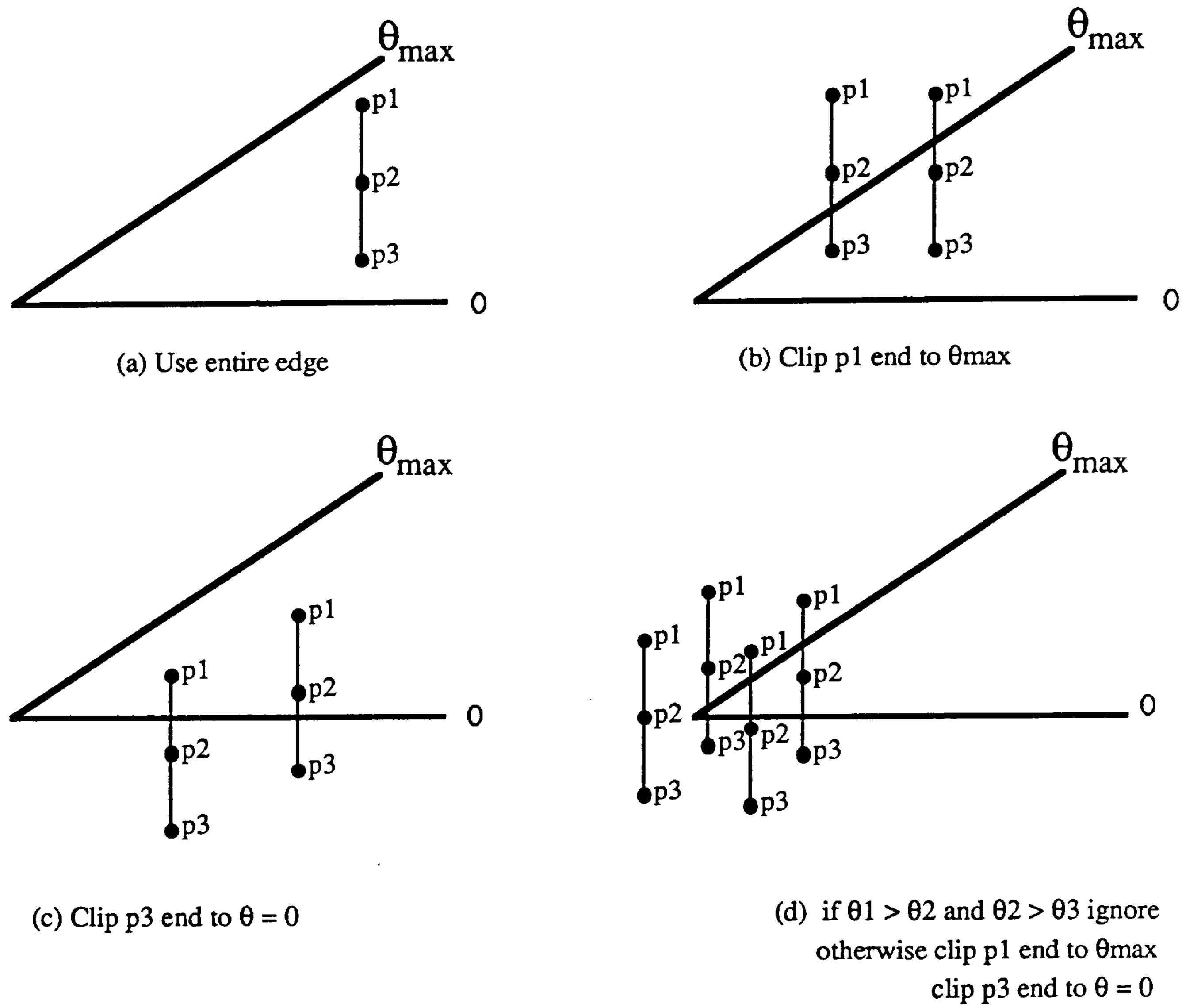


Figure 5. The possible clipping arrangements for a straight line edge intersecting a cylindrical wedge.

Solving for the intersection point with the $\theta = \theta_{max}$ wedge boundary is straightforward using the same basic approach and a simple trick. If the coordinates of the edge are rotated about the z-axis by negative θ_{max} radians, then the y-coordinate of the intersection point (b above) becomes zero.

To do this, the vector d needs to be changed to $d = [-sin(\theta_{max})\ cos(\theta_{max})\ 0]$ and the above calculations repeated [Hearn, pp. 108-109].

Since it is already known that the edge under study intersects the wedge boundary (due to previous calculations), the above solution holds unless p_1 and p_2 are identical.

3.2.1.4 Closing Faces Against the Rotational Sweep Wedge

In the previous section, two different methods are presented which demonstrate how an edge can be clipped against the constant θ boundaries of the sweep wedge. In the first method, the edges are clipped against each planar boundary separately. Closure of each face along the clipping plane is routine provided that the edges bounding the face are processed in an orderly fashion [Hearn, pp. 134-138].

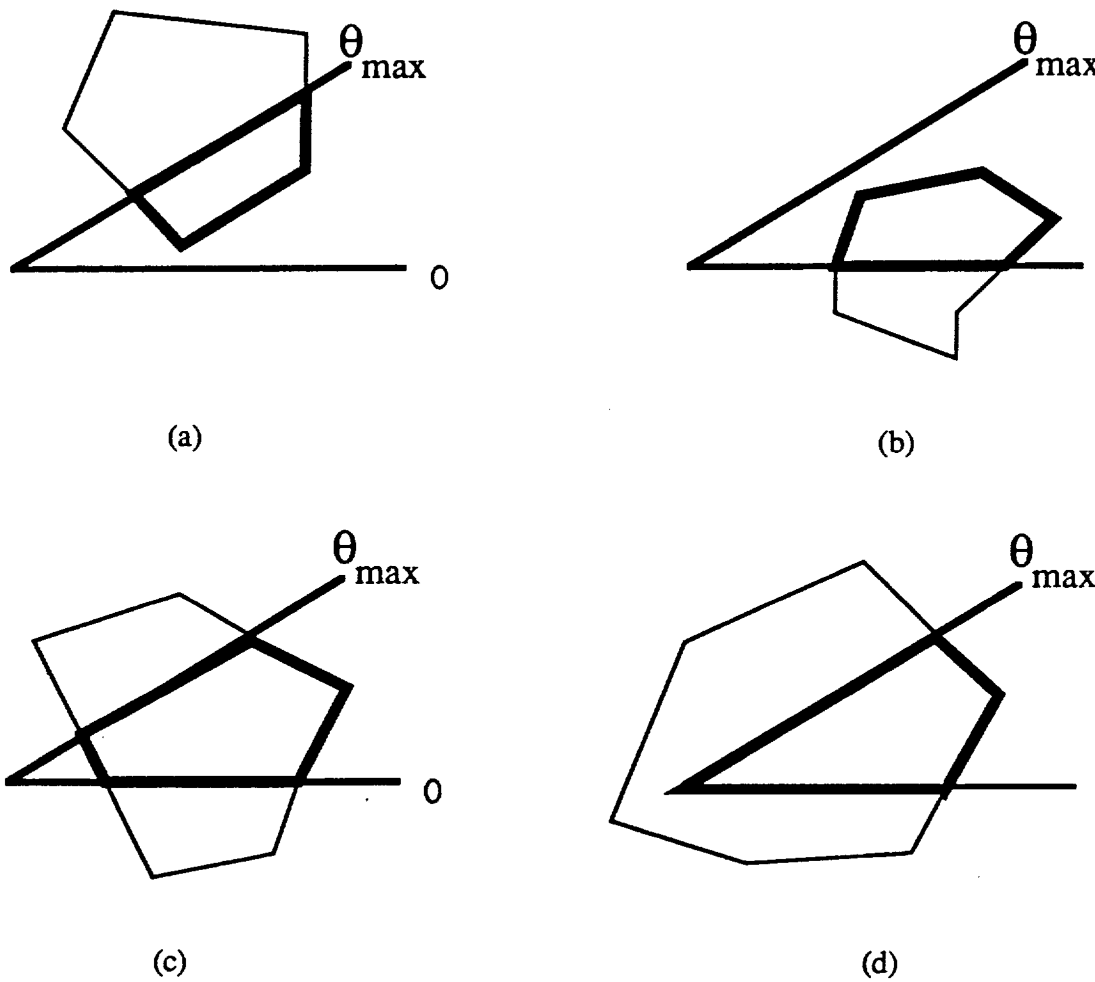


Figure 6. The results of clipping and closing a polygonal face against a cylindrical wedge. The heaviest lines represent the final shape of the face.

For the second method where clipping against both boundaries is done simultaneously, the results of clipping and the necessary closure are not as clear cut. Figure 6 shows the four types of closure possible for the positive radius wedge. In Figure 6a and 6b the method of closure is identical to that used when clipping a face against a single plane. Figure 6c shows a polygon which is closed against both faces and Figure 6d depicts a polygon that leaves from one of the wedge boundaries and enters via the other. In this case, the resultant closure requires the addition of two edges instead of the more typical one.

Unlike the normal closure case, the clipping algorithm does not supply all the information necessary to construct the two new edges of Figure 6d. The data missing is the value of the z or height, h, of the face where the radius is zero. According to calculus, a plane can be defined by a normal vector to the plane and a point on the plane by the function:

$$(p - p_0)\cdot n = 0$$

where p_0 is the given point on the plane and n is the normal to the plane. Knowing that the radius must be zero, and hence both x and y must be zero, reduces the formula to:

$$[\langle 0\ 0\ h\rangle - \langle x_0\ y_0\ z_0\rangle]\cdot[n_x\ n_y\ n_z] = 0$$

and

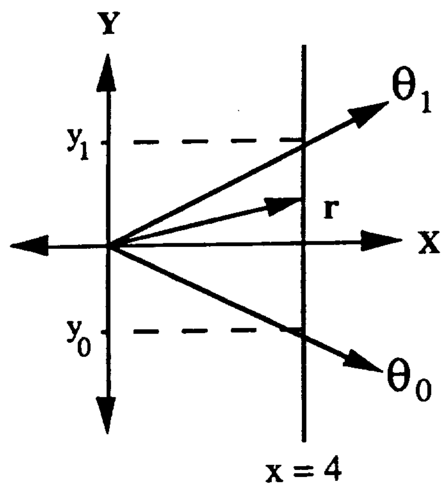
$$h = x_0 \frac{n_x}{n_z} + y_0 \frac{n_y}{n_z} + z_0$$

the clipping and closure algorithms guarantee that the face crosses the radius-equals-zero point, the value of n_z must be nonzero.

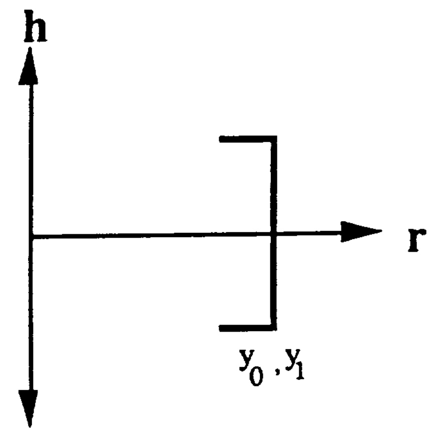
3.2.2 Discovered Inadequacies of the Current Solution

At first glance it would appear that all the necessary elements are now in place to generate the complete rotational sweep shadow for a group of objects and an arbitrary rotational reference. Unfortunately, this is not the case. There are two main problems which need to be addressed. The first is caused by values of the rotation angle greater than π radians. The reason that this is a problem is that when the sweep plane is rotated greater than π radians it becomes possible for the same points on an object to intersect both the positive and negative radius sides of the sweep plane.

The second problem is illustrated in Figure 7. Consider the planar face represented by the plane $x = 4$ as shown in Figure 7a. If the face is bounded at height of $z = \pm 2$ and the algorithms presented above are used to sweep from θ_0 to θ_1 the resulting plot in the rh-plane will result as shown in Figure 7b. The reason that the left side of the figure is left open is due to the manner in which the wireframe is swept. As the $z = 2$ edge is swept, the h value of the edge is constant and the r value starts at some value greater than 4 ($= \sqrt{(y_0 * y_0 + 4*4)}$), decreases to 4 at $y = 0$ and then increases again as θ reaches θ_1. When one of the edges at the constant θ boundaries is determined (via clipping and closure) a constant radius edge is generated in the rh-plane. Similar results occur at the $z = -2$ and θ_0 boundaries respectively. For the case when y_0 and y_1 are the same angular distance from the x-axis, the open ended wireframe of Figure 7b will be generated. This is called the minimal radius edge problem and is solved below.



(a) A planar face at x = 4



(b) The results of clipping and closing the wireframe of part (a).

Figure 7. An illustration of the minimal radius edge problem.

3.2.2.1 Dealing with Rotational Angles Greater Than π

There are two ways of dealing with rotational angles greater than π. If the first clipping algorithm is used (clipping versus the boundaries separately) then some modifications need to made in the way that edges are clipped by the planes representing the wedge boundaries. Two potential types of modifications would work in this case. The first would be to institute a clip against a half-plane using the origin as the boundary of the half-plane and then clipping against the two half-planes which represent the wedge boundaries. The second technique would involve clipping against the two full-planes representing the extended wedge boundaries and performing a union of the resulting edges. Neither of these options is very desirable, in the first clipping against a half-plane is not well defined and in the second the union operation can be computationally expensive.

When simultaneous clipping against both the wedge boundaries is used, the options for solving this problem are not straightforward. The added difficulty here is that it is now possible for an edge to have both end-points in the unclipped zone and yet still have a portion of itself in the clipped zone. When this is the case, two possible configurations are possible. The first is that the edge should be closed via the radius-equals-zero point; a problem that is already solved. The second configuration describes the situation when the clipped wedge cuts the object into two sections thus causing the face under study to become two faces. In this case, each face with this configuration would have to be clipped and closed twice, once for each side.

If it is only important to know the boundaries of the sweep shadow and extra lines internal to the shadow itself are not important or are being removed in a later step, then the following technique can be used for both independent and simultaneous clipping operations. Since it is demonstrated in previous sections how to clip and close when the rotation angle is less than or equal to π, this problem can be divided into sweeps with angles less than or equal to π. The most obvious choice of divisions is for one sweep with an angle of π and the other with a sweep of θ_{max} minus π. This requires two extra passes of each face (for a total of four) through the clipping and closure routines, but it will determine the full boundary of the sweep shadow. One important consideration is to keep track of whether the clipped edges have a positive or negative radius since the second sweep wedge begins at angle π and not at angle zero.

3.2.2.2 Solving the Minimal Radius Edge Problem

The problem illustrated in Figure 7 is not as simple to solve as that of rotation angles greater than π. The problem stems from the fact that the wireframe of an object's faces does not necessarily define the radial boundaries of those faces. By inspection it can be seen that the maximum radius of any point on a plane is infinite, therefore, the wireframe boundary of a face on that plane will define the maximum radii of the points on that face. The wireframe, though, does not always specify the minimal radii points on that face and this is the situation depicted in Figure 7. Thus, it is necessary to locate and sweep the minimal radius boundary for each face.

A planar surface represented in cylindrical coordinates is not simply defined. A brief look at the mathematics, though, reveals that the problem is not difficult to solve. Start with the basic equation for a plane.

$$Ax + By + Cz = D$$

By fixing the value of z and solving for the line on the plane defined by this fixed z yields:

$$y = -\frac{A}{B}x + \frac{D - Cz}{B}$$

Use this to compute the radius squared, and take the partial derivative of this square with respect to x and set it equal to zero. This describes the point on that line where the radius is minimized.

$$r^2 = x^2 + y^2 = x^2 + (\frac{A}{B}x + \frac{D - Cz}{B})^2$$

$$\frac{\partial r^2}{\partial x} = 2x(1 + \frac{A^2}{B^2}) - 2\frac{A}{B^2}(D - Cz) = 0$$

$$x = \frac{A}{(A^2 + B^2)}(D - Cz)$$

By allowing z to vary, a line representing the minimum radius edge of the plane is defined. This line can then be treated like any other edge on the face.

There are a few special cases for the proceeding calculations. For the case when $C = 0$, the values for x and y are constant throughout the plane. When both $A = B = 0$ this is the case where $z = D$ and the plane has a minimal radius of zero which is captured by the closure algorithm and can be ignored. For the final special case when $B = 0$, the normal of the plane of the face has no y component and the value of zero for y can be used.

Now that the minimum radius line for the plane representing the face is defined some further processing needs to be performed. First, it must be determined whether the line even intersects the area of the face. This is just the Polygon Intersected Edges problem as defined by Preparata and Shamos [Prep, p. 313]. If the line does intersect the polygon defining the face then extra edges must be added where appropriate. (This is one edge for a convex polygon and possibly more for a non-convex polygon.)

A solution to this problem is to extend the single-shot polygon inclusion test so that all the desired edges are obtained [Prep, pp. 41-43]. The basic idea of the single-shot polygon inclusion test is to determine whether a point is located inside a polygon by drawing a ray from that point to infinity and counting the number of intersections with the edges of the polygon. An odd number of intersections means the point is inside and an even number means the point is outside. The minimum radius line is used to define the direction of this ray. Rays are drawn in both directions away from the point and by sorting all the intersection points, a set of minimal radius edges can be determined by alternately labeling each intersection point with the rays as inside or outside the polygon defining the face.

There are a few non-trivial problems associated with this approach which are only briefly mentioned by Preparata and Shamos. These involve degenerate intersections between the line and the edges of the polygon. The first arises when the line intersects the polygon at a vertex. Not only does this intersection point intersect two edges, it is also quite possible that the line is only grazing the polygon and does not enter or leave it at that point. Whether this type of intersection represents a true intersection with the polygon can be determined by looking at the sign of the sine of the angle between the line and the edges of the polygon when both of the edges are directed in the same direction about the polygon. This information is readily determined by using the vector cross product.

The second problem arises when the line and an edge overlap each other. This, though, is just an extension of the vertex intersection problem. By looking at the edges on both ends of the overlapping edge it can be determined whether this type of intersection defines a crossing point or not.

3.2.3 The Complete Solution

In Figure 3, a simplified algorithm is provided which attempts to solve the rotational sweep shadow problem. In previous sections, this algorithm is shown to be mostly correct, but lacks certain features which leave the problem incompletely solved. Figure 8 depicts an enhanced version of the algorithm in Figure 3 with added steps for solving the problems associated with large sweep angles and minimal radius edges.

Even though the solution to this problem requires up to four passes through the clipping and closure routines for each wedge, the time complexity is still linear with respect to the number of edges in the obstacles. The only portion of the solution which is not linear is the sorting of intersection points found in the minimal radius edge solution. In the worst case this sorting can be done in nlogn time, but it is more likely that only zero, two, or four points need be sorted for a face with most faces possessing zero or two intersections for reasonable objects.

3.2.4 Results

A rotational sweep shadow generator using the techniques of simultaneous clipping as described above has been implemented in C on a Sun-3 workstation. It is incorporated within a robotic assembly planning system and is used to analyze mating trajectories, grasp sites, and grasp types for the case of simple rotations. Figure 9 shows the results of this algorithm as it encounters a cube. In both the cases, the extra internal lines of the shadow are left to aid in visualization and because their presence does not affect the operation and results of the larger system.

Figure 9a shows the cube when the center of the rotation lies within the cube. The rotation angle is 2 radians and the apparent extra set of lines internal to the right and left sides of the figure result from the minimal radius edge calculations. Figure 9b shows then same cube when the center of the rotation lies outside of the cube. The triangular patch to the right shows the extra face caused by

```
procedure GENERATE_COMPLETE_ROTATIONAL_SWEEP_SHADOW;
    specify and define the cylindrical coordinate system for the sweep;
    for each object in the environment;
        for each face in the environment;
            find the height of the face at r=0;
            find the minimal radius edges;
            for each minimal radius edge;
                if θmax > π then;
                    sweep in two parts;
                else;
                    sweep in one part;
                end-if;
            for each edge on the face;
                convert the edge to the sweep coordinate system;
            if θmax > π then;
                for each edge on the face;
                    sweep for θ < π and clip as necessary;
                    close with the previous edge if necessary;
                close the face between the initial and final vertices;
                reduce θmax by π;
                rotate all the data π radians;
            end-if;
            for each edge on the face;
                sweep the edge and clip as necessary;
                close with the previous edge if necessary;
            close the face between the initial and final vertices;
end-procedure;
```

Figure 8. A complete algorithm for solving the rotational sweep shadow problem.

clipping and closure at one of the wedge boundaries. The two vertical lines which are tangent to two of the curved edges are each minimal radius edges. In the case of the right-most one, its presence in necessary.

4.0 Analyzing the Sweep Shadow

Since volume sweeping is equivalent to sweeping areas within a plane, it is necessary to determine how much cross-sectional area remains for the gripper and to employ a metric which characterizes this area. Because this value depends heavily on the actual location of the grasp sites, the grasp sites need to be known at this time. A list of grasp sites and types can be provided or some type of grasp planner may be used.

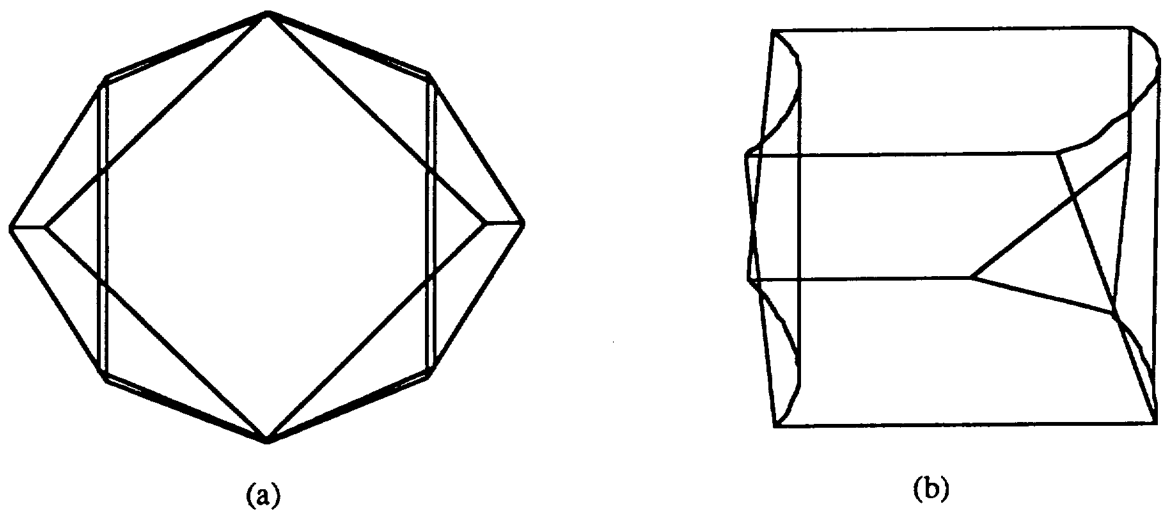


Figure 9. The rotational sweep shadow of a cube. (a) With the center of rotation inside the cube. (b) With the center of rotation outside of the cube.

Once the grasp sites are known, their locations are mapped to the sweep plane. Based on their type, (e.g., one-fingered) the unswept areas in the plane are scanned. Figure 10 shows the area which must be considered for most of the grasp sites and types, since the obstructions in all directions around the grasp site may be of importance. In the case where it is known that the gripper extends in a direction perpendicular to that of the motion (e.g., a two-fingered grasp from above during a horizontal motion) only half of the unswept plane needs to be considered.

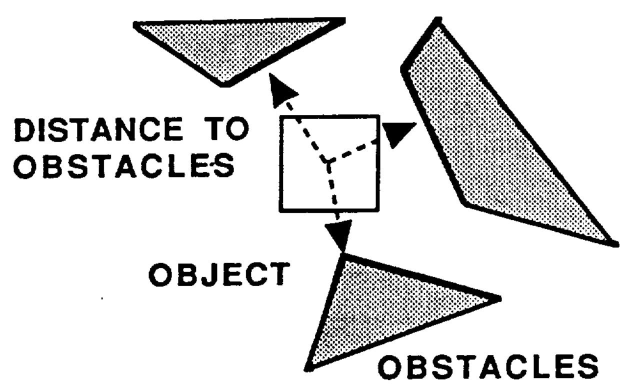


Figure 10. Use of distance to the nearest obstacle as a measure of a gripper's envelope.

Once the appropriate unswept areas in the plane have been determined, it is necessary to analyze them and to compute the value of a metric on their suitability. A first approach might to find the largest rectangle which will fit in the unswept area, while still maintaining a reasonable aspect ratio. Doing this is not completely straight forward because the rectangle needs to be roughly centered about the grasp site's projection onto the plane. It is also necessary to incorporate a polygon fitting algorithm to find the valid positions and sizes of the gripper rectangle.

This relatively complex procedure can be replaced by a simpler procedure which effectively approximates the unswept area. By measuring the euclidean distance from the center of the grasp site to the nearest obstruction, an inscribing circular (or semicircular) area is defined within the unswept volume. By definition, this area is completely free of obstacles and its radius is an accurate measure of its size. Figure 11a depicts an example of the calculation of this metric. While this method does not always rate the unswept area as well as a well-fit rectangle might, as shown in Figure 11b, the inscribed circular area gives a conservative estimate of the cross-sectional area available for the gripper. This technique is also more computationally efficient than one which fits a rectangle into the unswept area.

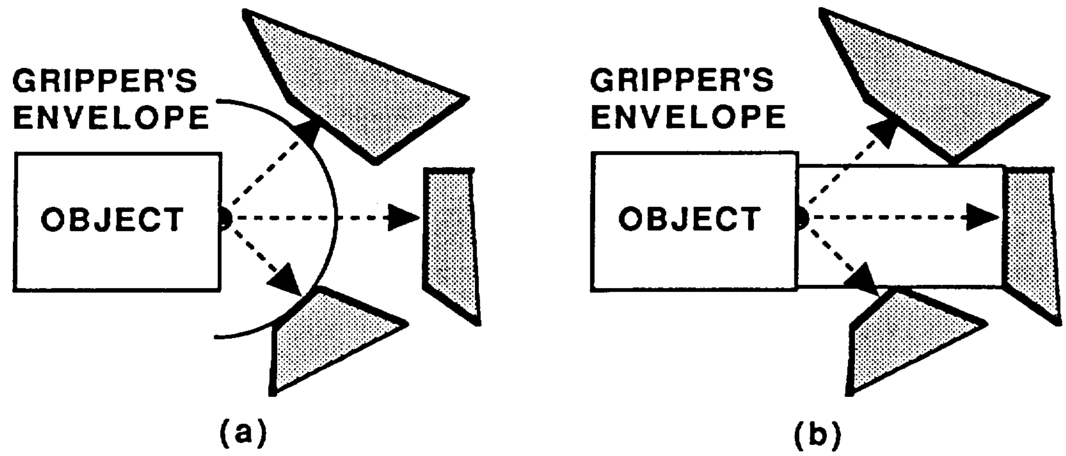


Figure 11. Difference between (a) the inscribed circle and (b) the fit rectangle measures.

5.0 The Complete Plane Sweep Algorithm

A step by step synthesis of an algorithm to solve the gripper envelope problem has been presented. A pseudocode description of the algorithm is provided as Figure 12.

The complexity of each of the components in the algorithm is presented in Table 1.

Table 1 - Complexities for the Plane Sweep Algorithm

Component	Complexity
Placement of Vertices	O(N)
Extraction of Vertices	O(N)
Plane Sweep Along a Straight Line	O(N)
Plane Sweep Through a Simple Rotation	O(N)
Halving of a Plane	O(N)
Finding Radius of Largest Inscribed Circle	O(N)
Complete Algorithm	O(M*N*L)

where N is the number of vertices in the obstructing objects,
M is the number of trajectories, and
L is the number of grasp sites per trajectory.

```
procedure ANALYZE_GRIPPER_ENVELOPE;
        place the vertices of all the obstacles in 3-D space;
        extract the vertices of the obstacles to be considered;
        for each mating trajectory;
            for each trajectory segment;
                if segment type is straight line segment then;
                    for each edge in the obstacles;
                        project and clip the edge rectangularly;
                else;
                    for each edge in the obstacles;
                        project and clip the edge cylindrically;
                end-if;
                for each possible grasp site;
                    if full plane type then;
                        find the distance to the nearest obstacle;
                    else;
                        discard half the plane;
                        find the distance to the nearest obstacle;
                    end-if;
                    record the distance found above;
    end-procedure;
```

Figure 12. Pseudocode description of the Plane Sweep algorithm.

6.0 Conclusion

The plane sweep algorithm has been implemented in C on a Sun-3 workstation. The appendices provide some of the results obtained from the algorithm for the class of objects restricted to those with planar surfaces only and the mating trajectories restricted to sequences of straight line segments and simple rotations.

These test results demonstrate that the plane sweep algorithm's solution to the gripper envelope problem provides a satisfactory rating of various assembly sequences. Its ability to analyze and approximately measure the amount of volume available for an unspecified gripper, given different mating trajectories, grasp sites and gripper types, makes it useful as an aid in rating and ranking assembly sequences.

7.0 Future Research

There are many areas in which the plane sweep algorithm can be enhanced. More complicated trajectories can be allowed, such as helical straight line translations or even appropriately curved trajectories. It is only necessary to properly parameterize the environment with respect to the direction of motion. More complex shapes can also be allowed in the form of cylindrical or spherical subvolumes. It is also be possible to place some form of cost function on points in the unswept areas, to give an indication of a gripper angled more in one direction than another. Other assembly factors can be integrated with this algorithm within the assembly sequence planner to give a better overall picture as to which assembly sequence may be the best one available.

Research can be performed on other possible applications of an algorithm of this type. Some of the ideas that come to mind are: applications in obstacle avoidance, motion planning, and determination of the amount of tracking error allowable during an actual motion.

Figure 13 depicts a mobile robot path planning problem involving the choice of paths either around or through a set of obstacles. The plane sweep algorithm is useful in determining the size of the obstacle free area about the robot. By combining the straight line trajectories with the simple rotations at the trajectory vertices, a very accurate indication of the free space is determined. The plane sweep algorithm, however, still predicts that the best trajectory is the one avoiding the obstacles since it is safer. To take into account the effects of trajectory length, a heuristic function must be implemented which balances the safety issues inherent in the gripper envelope metric with the mating time issues implied by the trajectory length.

Acknowledgement

This work was supported in part through NASA grant NAGW-1333.

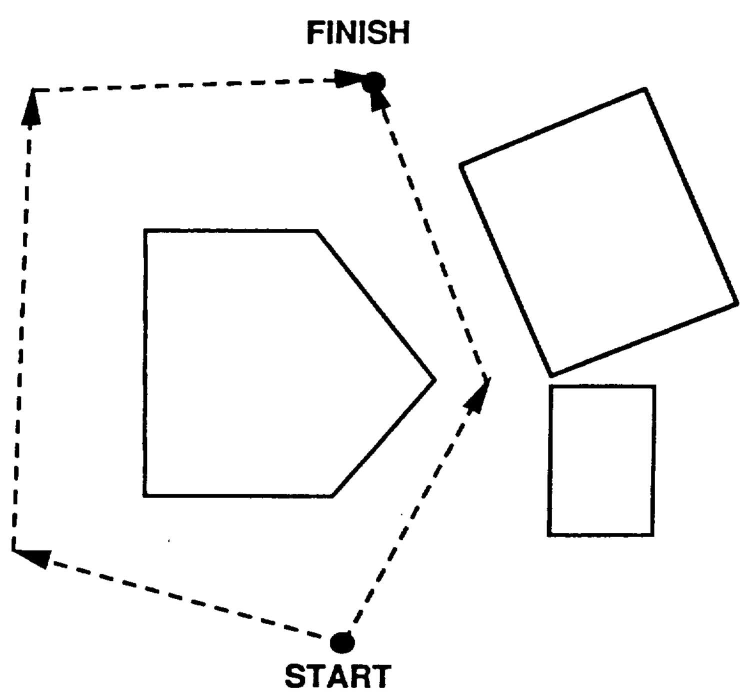


Figure 13. Example test case for ranking trajectories. A maze navigation problem with multiple trajectories.

BIBLIOGRAPHY

[Blinn] Blinn, J. F., "Optimal Tubes," *IEEE Computer Graphics and Applications*, September 1989, pp. 8-13.

[De Fazio87] De Fazio, T. L. and Whitney, D. E., "Simplified Generation of all Mechanical Assembly Sequences," *IEEE Journal of Robotics and Automation*, December, 1987, pp. 640-658.

[De Fazio88] De Fazio, T. L. and Whitney, D. E., "Correction to 'Simplified Generation of all Mechanical Assembly Sequences'," *IEEE Journal of Robotics and Automation*, December, 1988, pp. 705-708.

[Foley] Foley, T. A., et al., "Visualizing Functions Over a Sphere," *IEEE Computer Graphics and Applications*, January, 1990, pp. 32-41.

[Hearn] Hearn, D. S. and Baker, M. P., *Computer Graphics*, Prentice Hall, Englewood Cliffs, NJ, 1986.

[Hoffman] Hoffman, K. and Kunze, R., *Linear Algebra*, Prentice Hall, Englewood Cliffs, NJ, 1971.

[Homem86] Homem de Mello, L. S. and Sanderson, A. C., "AND/OR Graph Representation of Assembly Plans," *AAAI-86 Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986, pp. 1113-1119.

[Homem88] Homem de Mello, L. S. and Sanderson, A. C., "Task Sequence Planning for Assembly," *IMACS World Congress '88 on Scientific Computation*, July 1988.

[Ko] Ko, H. and Lee, K., "Automated Assembling Procedure Generation from Mating Conditions," *Computer Aided Design*, January-February, 1987, pp. 3-10.

[Mirolo] Mirolo, C. and Pagello, E., "A Solid Modeling System for Robot Action Planning," *IEEE Computer Graphics and Applications*, January, 1989, pp. 55-69.

[Prep] Preparata, F. P. and Shamos, M. I., *Computational Geometry an Introduction*, Springer-Verlag, New York, NY, 1985.

[Selby] Selby, S. M., ed., *Standard Mathematical Tables*, The Chemical Rubber Co., Cleveland, OH, 19th. Ed., 1971.

[Turner] Turner, J. E., "Accurate Solid Modeling Using Polyhedral Approximations," *IEEE Computer Graphics and Applications*, May, 1988, pp. 14-28.

[Wang] Wang, W. P. and Wang, K. K., "Geometric Modeling for Swept Volume of Moving Solids," *IEEE Computer Graphics and Applications*, December, 1986, pp. 8-17.

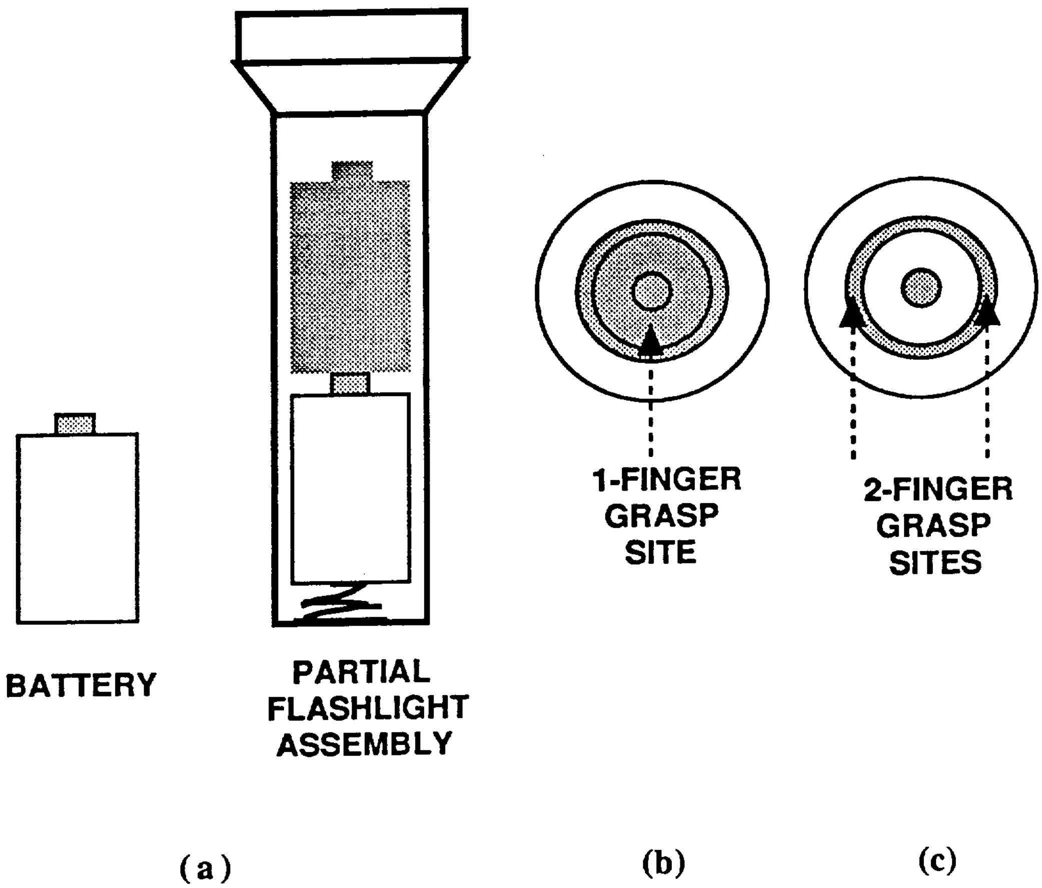
[Welch] Welch, Henry L., *Robot Independent Assembly Sequence Planning*, PhD. Thesis, Rensselaer Polytechnic Institute, August 1990.

Appendix A
Sample Test Cases

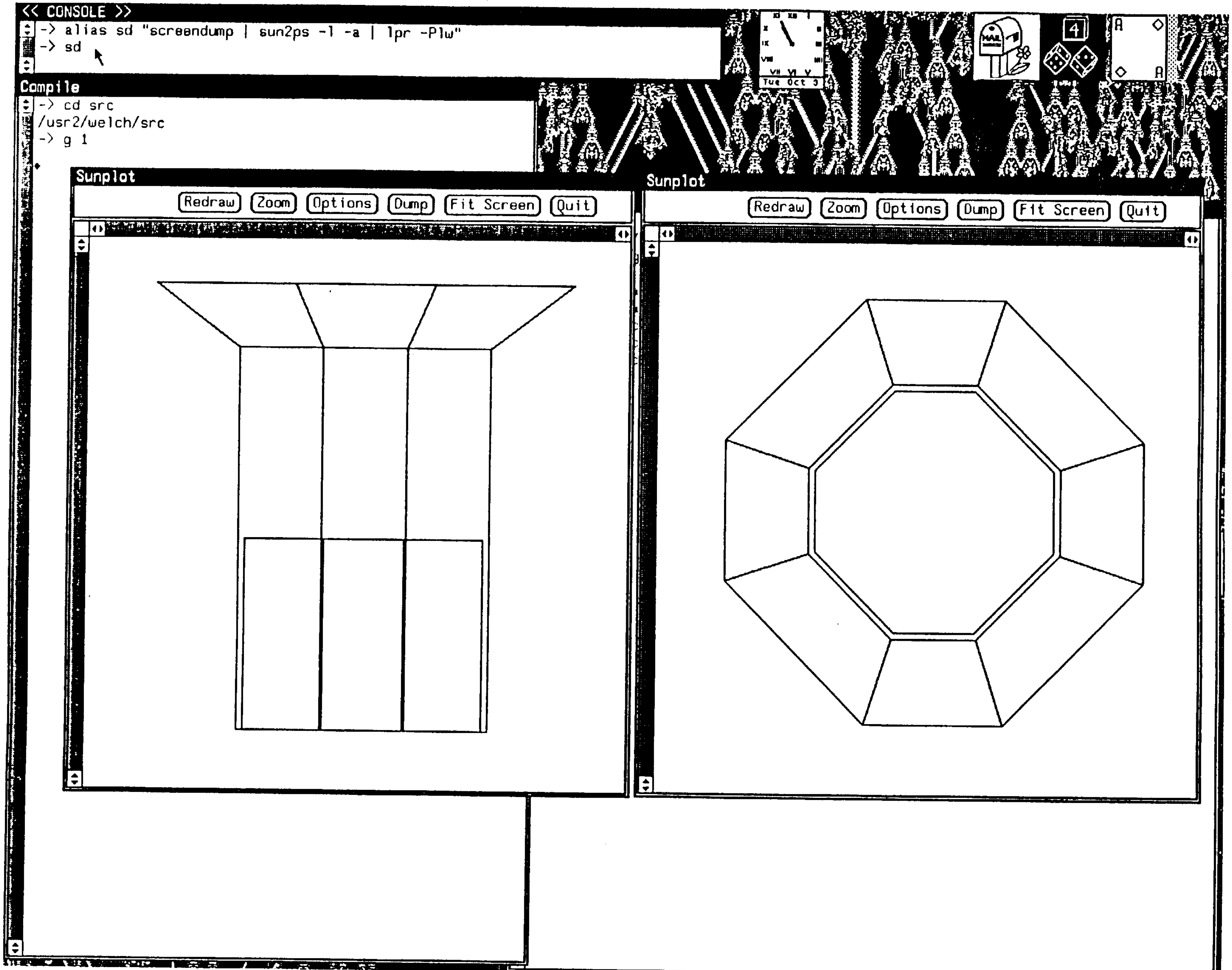
Case 1: The 2 D-Cell Flashlight

This test case is used to demonstrate the plane sweep algorithm's ability to choose more appropriate grasp types.

Associated Files: ~/case1.dat
~/case1.trj



Case 1. (a) Stylized two-cell flashlight with one battery installed.Grasp sites available for (b) one-fingered gripper and (c) two-fingered gripper.



<< CONSOLE >>
-> alias sd "screendump | sun2ps -l -a | lpr -Plw"
-> sd
Compile
-> cd src
/usr2/welch/src
-> g 1
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit

```
2
24      17
0.0     0.0     1.0
0.0     0.0     0.0
0.5     1.5     -3.5
1.5     0.5     -3.5
1.5     -0.5    -3.5
0.5     -1.5    -3.5
-0.5    -1.5    -3.5
-1.5    -0.5    -3.5
-1.5    0.5     -3.5
-0.5    1.5     -3.5
0.5     1.5     2.5
1.5     0.5     2.5
1.5     -0.5    2.5
0.5     -1.5    2.5
-0.5    -1.5    2.5
-1.5    -0.5    2.5
-1.5    0.5     2.5
-0.5    1.5     2.5
0.833   2.5     3.5
2.5     0.833   3.5
2.5     -0.833  3.5
0.833   -2.5    3.5
-0.833  -2.5    3.5
-2.5    -0.833  3.5
-2.5    0.833   3.5
-0.833  2.5     3.5
8       0       1       2       3       4       5       6       7
        0.0     0.0     -1.0
4       0       1       9       8
        0.707   0.707   0.0
4       1       2       10      9
        1.0     0.0     0.0
4       2       3       11      10
        0.707   -0.707  0.0
4       3       4       12      11
        0.0     -1.0    0.0
4       4       5       13      12
        -0.707  -0.707  0.0
4       5       6       14      13
        -1.0    0.0     0.0
4       6       7       15      14
        -0.707  0.707   0.0
4       7       0       8       15
        0.0     1.0     0.0
4       8       9       17      16
        0.515   0.515   -0.686
4       9       10      18      17
        0.707   0.0     -0.707
4       10      11      19      18
        0.515   -0.515  -0.6867
4       11      12      20      19
        0.0     -0.707  -0.707
4       12      13      21      20
        -0.515  -0.515  -0.686
4       13      14      22      21
        -0.707  0.0     -0.707
4       14      15      23      22
        -0.515  0.515   -0.686
4       15      8       16      23
        0.0     0.707   -0.707
16      10
0.0     0.0     1.0
```

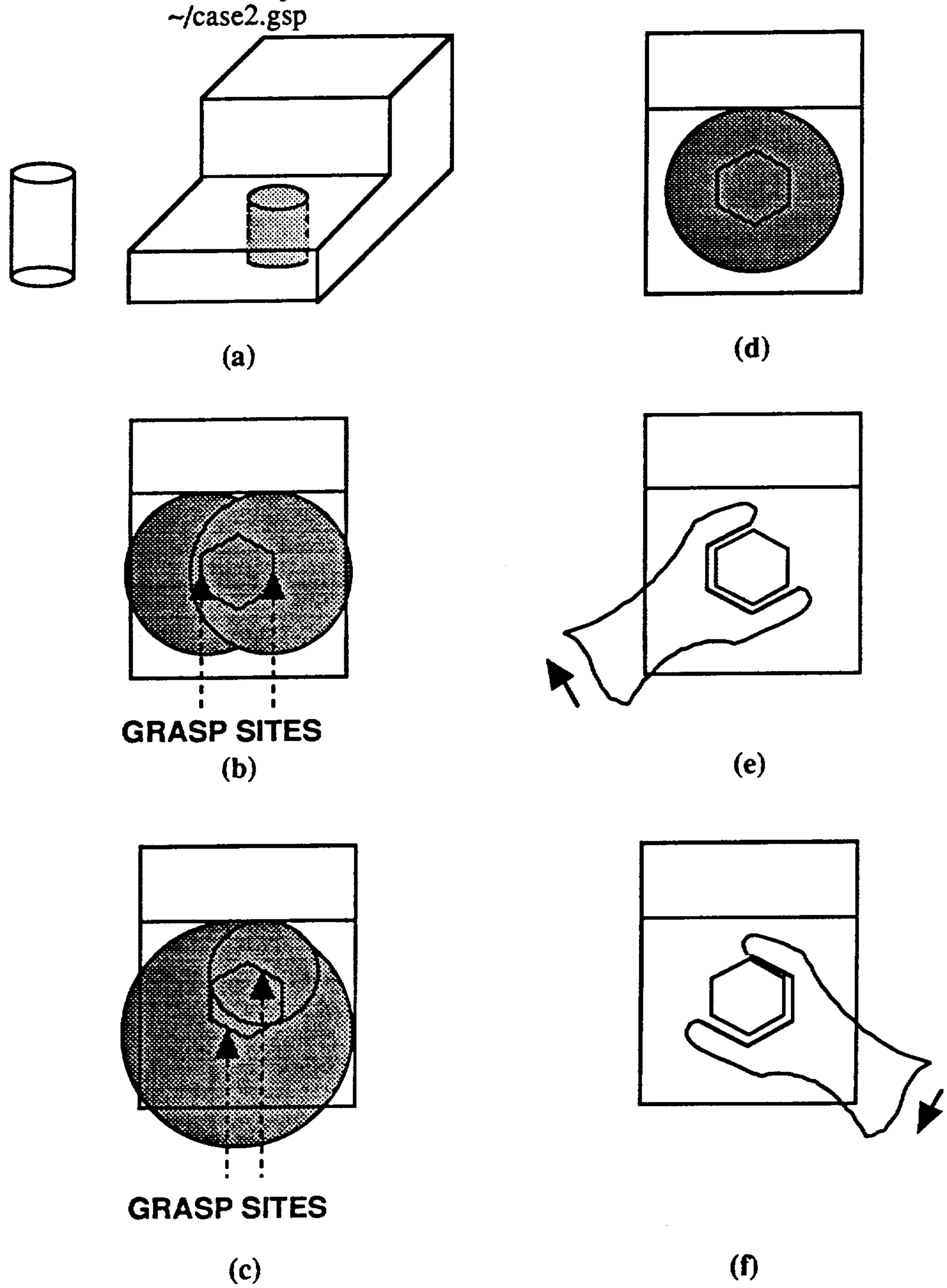
0.0	0.0	−2.0						
0.475	1.425	−1.5						
1.425	0.475	−1.5						
1.425	−0.475	−1.5						
0.475	−1.425	−1.5						
−0.475	−1.425	−1.5						
−1.425	−0.475	−1.5						
−1.425	0.475	−1.5						
−0.475	1.425	−1.5						
0.475	1.425	1.5						
1.425	0.475	1.5						
1.425	−0.475	1.5						
0.475	−1.425	1.5						
−0.475	−1.425	1.5						
−1.425	−0.475	1.5						
−1.425	0.475	1.5						
−0.475	1.425	1.5						
8	0	1	2	3	4	5	6	7
	0.0	0.0	−1.0					
4	0	1	9	8				
	0.707	0.707	0.0					
4	1	2	10	9				
	1.0	0.0	0.0					
4	2	3	11	10				
	0.707	−0.707	0.0					
4	3	4	12	11				
	0.0	−1.0	0.0					
4	4	5	13	12				
	−0.707	−0.707	0.0					
4	5	6	14	13				
	−1.0	0.0	0.0					
4	6	7	15	14				
	−0.707	0.707	0.0					
4	7	0	8	15				
	0.0	1.0	0.0					
8	8	9	10	11	12	13	14	15
	0.0	0.0	1.0					

```
1 0
1
0.0     0.0     5.75
0.0     0.0     -1.0
-1.0    0.0     0.0
0.0     -1.0    0.0
6.5
3
0       1.425   0.0
0       -1.425  0.0
0       0.0     0.0'-F
```

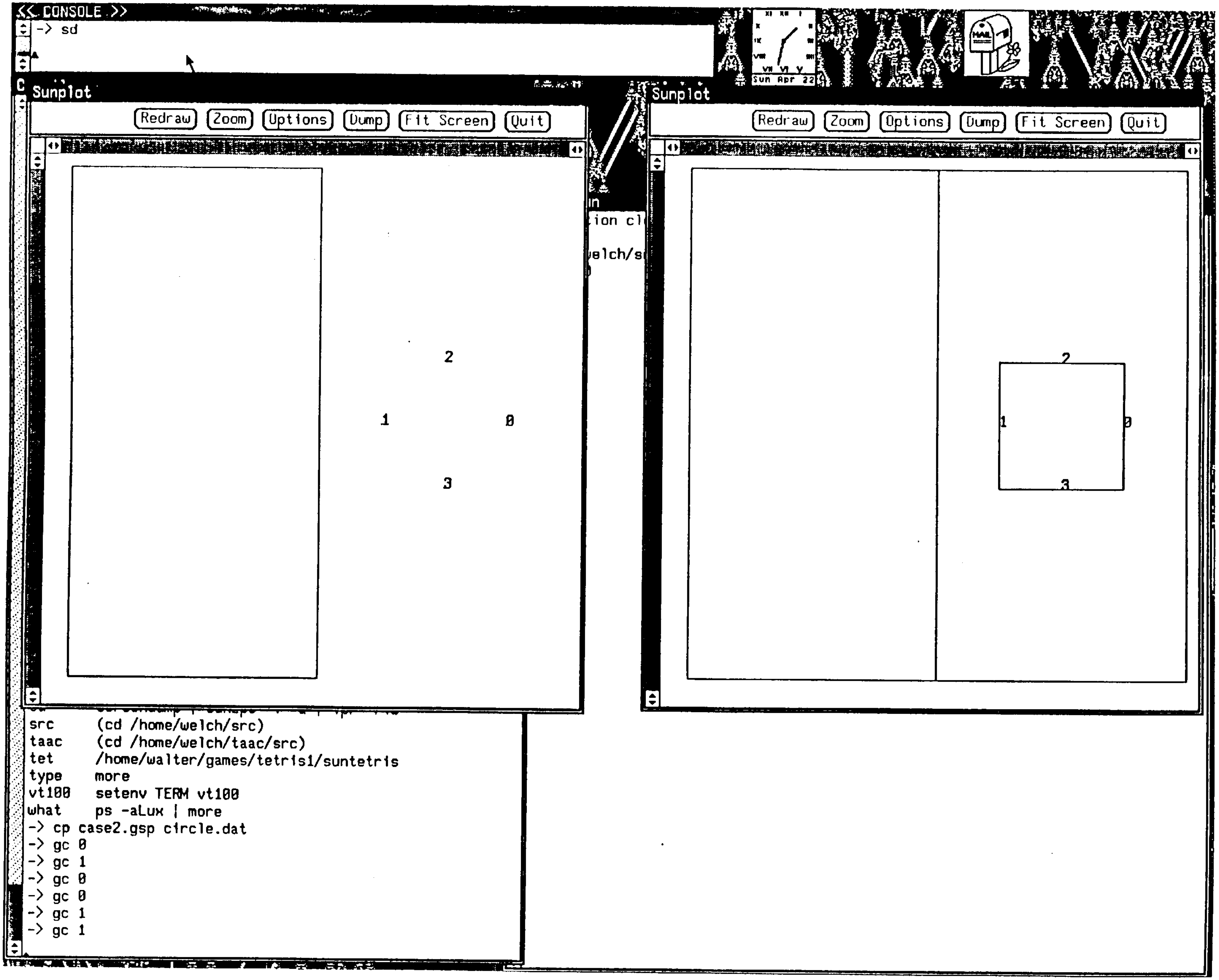
Case2: Peg-in-hole Assembly with Bounding Wall

This test case is used to demonstrate the plane sweep algorithm's ability to choose more appropriate grasp locations.

Associated Files: ~/case2.dat
~/case2.trj
~/case2.gsp



Case 2. (a) Peg in hole with side wall obstacle assembly. (b) and (c) The gripper envelope for a hex head bolt in place of the peg. (d) The gripper envelope for a socket attachment. (e) and (f) The effects of wrench placement which is detectable with a rotational sweep.


<< CONSOLE >>
-> sd
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
2
1
0
3
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
2
1
0
3
Sun Apr 22
src (cd /home/welch/src)
taac (cd /home/welch/taac/src)
tet /home/walter/games/tetris1/suntetris
type more
vt100 setenv TERM vt100
what ps -aLux | more
-> cp case2.gsp circle.dat
-> gc 0
-> gc 1
-> gc 0
-> gc 0
-> gc 1
-> gc 1

 case2.dat

```
2
20      12
0       0       1
0       0       0
0       2       0
0       2       1
2       2       1
2       2       3
4       2       3
4       2       0
0       -2      0
0       -2      1
2       -2      1
2       -2      3
4       -2      3
4       -2      0
1.5     -.5     0
1.5     .5      0
1.5     -.5     1
1.5     .5      1
4       0       6       11      5
        0       0       -1
6       0       1       2       3       4       5
        0       1       0
6       6       7       8       9       10      11
        0       -1      0
4       0       6       7       1
        -1      0       0
4       1       7       8       2
        0       0       1
4       2       8       9       3
        -1      0       0
4       3       9       10      4
        0       0       1
4       4       10      11      5
        1       0       0
4       12      16      17      13
        1       0       0
4       13      17      18      14
        0       -1      0
4       14      18      19      15
        -1      0       0
4       15      19      16      12
        0       1       0

8       6
0       0       .5
1       0       1
-1      -1      -2
-1      1       -2
1       1       -2
1       -1      -2
-1      -1      2
-1      1       2
1       1       2
1       -1      2
4       0       1       2       3
        0       0       -1
4       0       4       7       3
        0       -1      0
4       3       7       6       2
        1       0       0
4       2       6       5       1
        0       1       0
```

4	1	5	4	0
	−1	0	0	
4	4	5	6	7
	0	0	1	

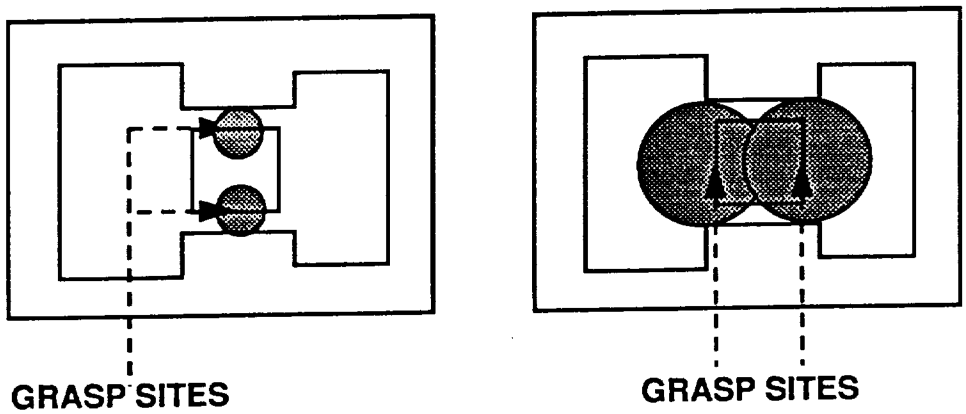
```
-.5 0 -.5 0 "1"
0 .5 0 .5 "2"
0 -.5 0 -.5 "3"
```

```
1
0
2
0
1 0 5
0 0 -1
-1 0 0
0 -1 0
4.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5

0
1 0 5
0 0 -1
-1 0 0
0 -1 0
3.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5
```

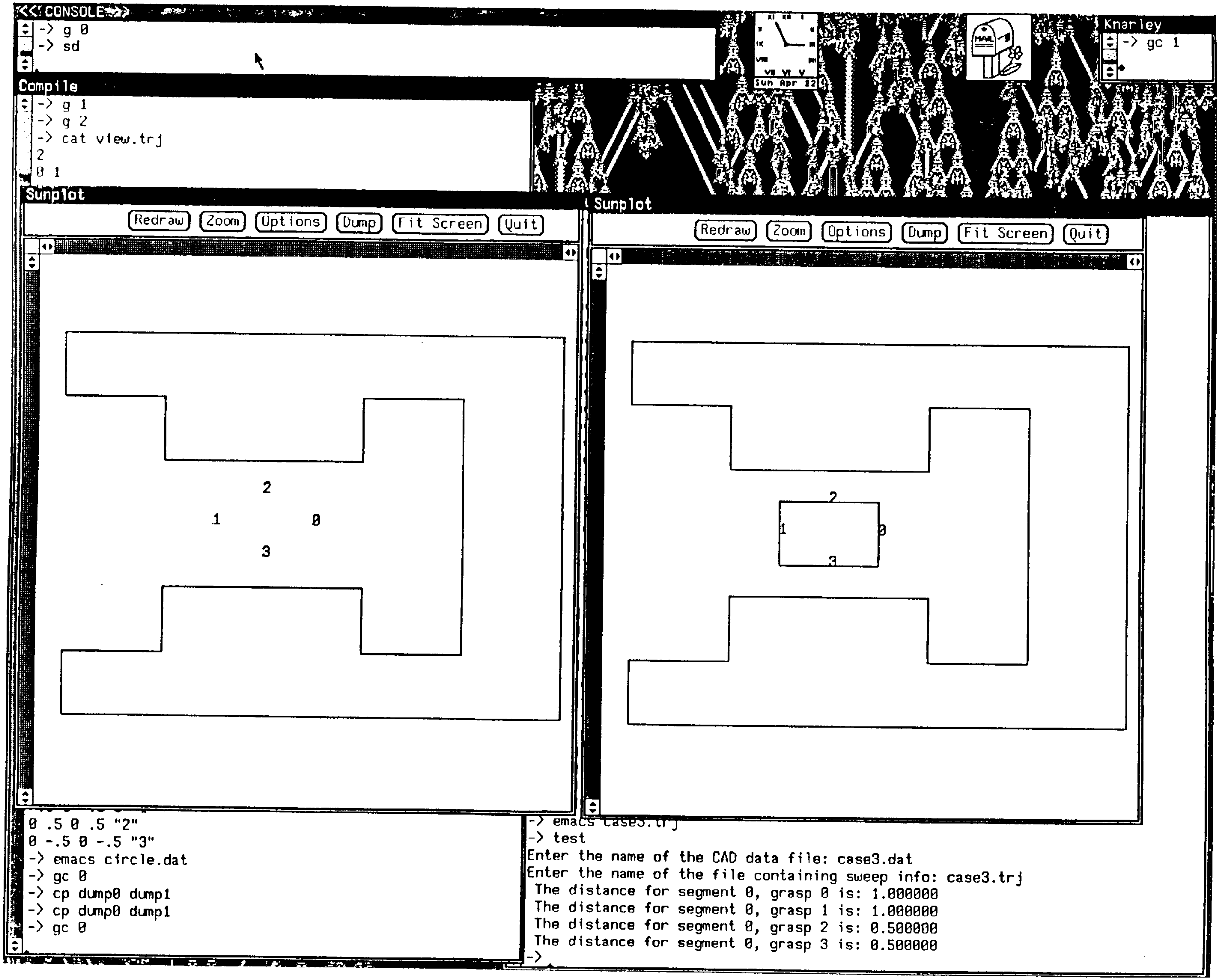
Case 3: Peg in a Pinched Box

This test case is used to demonstrate the plane sweep algorithm's ability to choose more appropriate grasp locations.

Associated Files: ~/case3.dat
~/case3.trj



Case 3. The results of applying the plane sweep to the objects of Figure 4. (a) The gripper envelope detected for the top-bottom grasp sites and (b) The gripper envelope detected for the side grasp sites.



CONSOLE
-> g 0
-> sd
Compile
-> g 1
-> g 2
-> cat view.trj
2
0 1
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
2
1
0
3
Sun Apr 22
Knarley
-> gc 1
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
2
1
0
3
0 .5 0 .5 "2"
0 -.5 0 -.5 "3"
-> emacs circle.dat
-> gc 0
-> cp dump0 dump1
-> cp dump0 dump1
-> gc 0
-> test
Enter the name of the CAD data file: case3.dat
Enter the name of the file containing sweep info: case3.trj
The distance for segment 0, grasp 0 is: 1.000000
The distance for segment 0, grasp 1 is: 1.000000
The distance for segment 0, grasp 2 is: 0.500000
The distance for segment 0, grasp 3 is: 0.500000
->

```
2
32  16
0          0          1
0          0          0
0 0 0
0 6 0
5 6 0
5 5 0
4 5 0
4 4 0
2 4 0
2 5 0
1 5 0
1 1 0
2 1 0
2 2 0
4 2 0
4 1 0
5 1 0
5 0 0
0 0 1
0 6 1
5 6 1
5 5 1
4 5 1
4 4 1
2 4 1
2 5 1
1 5 1
1 1 1
2 1 1
2 2 1
4 2 1
4 1 1
5 1 1
5 0 1
4 0 1 17 16        -1 0 0
4 1 2 18 17        0 1 0
4 2 3 19 18        1 0 0
4 3 4 20 19        0 -1 0
4 4 5 21 20        1 0 0
4 5 6 22 21        0 -1 0
4 6 7 23 22        -1 0 0
4 7 8 24 23        0 -1 0
4 8 9 25 24        1 0 0
4 9 10 26 25       0 1 0
4 10 11 27 26      -1 0 0
4 11 12 28 27      0 1 0
4 12 13 29 28      1 0 0
4 13 14 30 29      0 1 0
4 14 15 31 30      1 0 0
4 15 0 16 31       0 -1 0

8          6
0          0          .5
3          3          .5
-1         -1         -1
-1         1          -1
1          1          -1
1          -1         -1
-1         -1         1
-1         1          1
1          1          1
1          -1         1
```

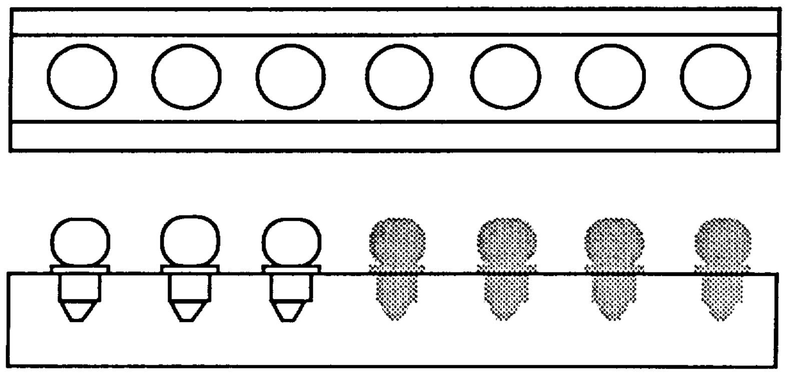
4	0	1	2	3
	0	0	−1	
4	0	4	7	3
	0	−1	0	
4	3	7	6	2
	1	0	0	
4	2	6	5	1
	0	1	0	
4	1	5	4	0
	−1	0	0	
4	4	5	6	7
	0	0	1	

```
1
0
1

0
3 3 3
0 0 -1
-1 0 0
0 -1 0
2.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5
```

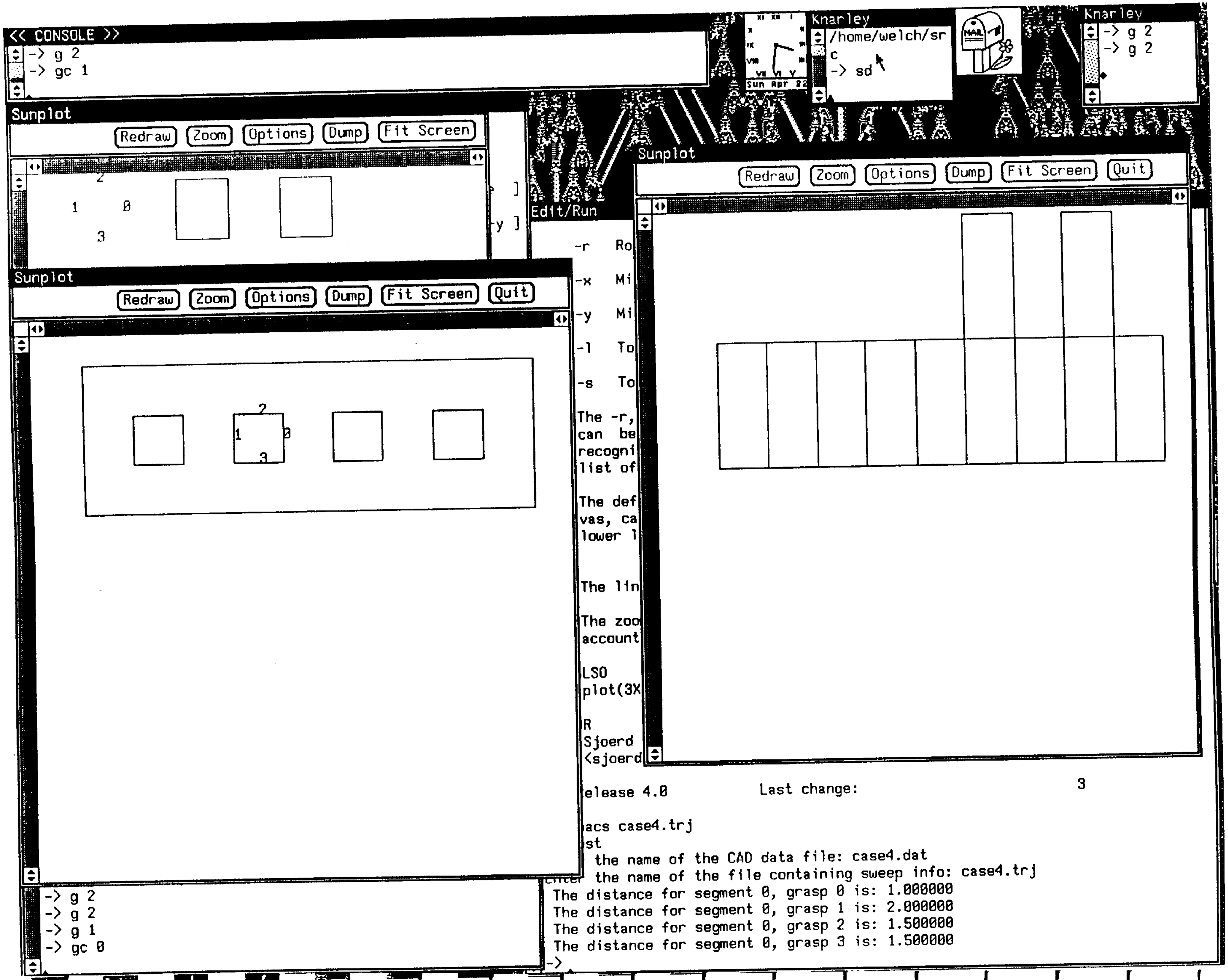
Case 4: Row of Indicator Lamps

This test case is used to demonstrate the plane sweep algorithm's ability to choose more appropriate grasp locations and assembly orders.

Associated Files: ~/case4.dat
~/case4.trj



Case 4. The row of lamps assembly. The presence of any lamps constrains the gripper space of the rest.



<< CONSOLE >>
-> g 2
-> gc 1
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
Knarley
/home/welch/sr
c
-> sd
-> g 2
-> g 2
Edit/Run
The -r,
can be
recogni
list of
The def
vas, ca
lower l
The lin
The zoo
account
LSO
plot(3X
Sjoerd
<sjoerd
elease 4.0
Last change:
acs case4.trj
the name of the CAD data file: case4.dat
the name of the file containing sweep info: case4.trj
The distance for segment 0, grasp 0 is: 1.000000
The distance for segment 0, grasp 1 is: 2.000000
The distance for segment 0, grasp 2 is: 1.500000
The distance for segment 0, grasp 3 is: 1.500000
-> g 2
-> g 2
-> g 1
-> gc 0

```
5

40 20
0 0 1
0 0 0
0 0 0
0 3 0
9 3 0
9 0 0
1 1 0
1 2 0
2 2 0
2 1 0
3 1 0
3 2 0
4 2 0
4 1 0
5 1 0
5 2 0
6 2 0
6 1 0
7 1 0
7 2 0
8 2 0
8 1 0
0 0 1
0 3 1
9 3 1
9 0 1
1 1 1
1 2 1
2 2 1
2 1 1
3 1 1
3 2 1
4 2 1
4 1 1
5 1 1
5 2 1
6 2 1
6 1 1
7 1 1
7 2 1
8 2 1
8 1 1
4 0 1 21 20      -1 0 0
4 1 2 22 21      0 1 0
4 2 3 23 22      1 0 0
4 3 0 20 23      0 -1 0
4 4 5 25 24      1 0 0
4 5 6 26 25      0 -1 0
4 6 7 27 26     -1 0 0
4 7 4 24 27      0 1 0
4 8 9 29 28      1 0 0
4 9 10 30 29     0 -1 0
4 10 11 31 30   -1 0 0
4 11 8 28 31     0 1 0
4 12 13 33 32    1 0 0
4 13 14 34 33    0 -1 0
4 14 15 35 34   -1 0 0
4 15 12 32 35    0 1 0
4 16 17 37 36    1 0 0
4 17 18 38 37    0 -1 0
4 18 19 39 38   -1 0 0
```

```
4  19  16  36  39       0  1  0

8           6
0           0           .5
1.5         1.5         1
-1          -1          -2
-1          1           -2
1           1           -2
1           -1          -2
-1          -1          2
-1          1           2
1           1           2
1           -1          2
4           0           1           2           3
            0           0           -1
4           0           4           7           3
            0           -1          0
4           3           7           6           2
            1           0           0
4           2           6           5           1
            0           1           0
4           1           5           4           0
            -1          0           0
4           4           5           6           7
            0           0           1

8           6
0           0           .5
3.5         1.5         1
-1          -1          -2
-1          1           -2
1           1           -2
1           -1          -2
-1          -1          2
-1          1           2
1           1           2
1           -1          2
4           0           1           2           3
            0           0           -1
4           0           4           7           3
            0           -1          0
4           3           7           6           2
            1           0           0
4           2           6           5           1
            0           1           0
4           1           5           4           0
            -1          0           0
4           4           5           6           7
            0           0           1

8           6
0           0           .5
5.5         1.5         1
-1          -1          -2
-1          1           -2
1           1           -2
1           -1          -2
-1          -1          2
-1          1           2
1           1           2
1           -1          2
4           0           1           2           3
            0           0           -1
4           0           4           7           3
```

	0	−1	0	
4	3	7	6	2
	1	0	0	
4	2	6	5	1
	0	1	0	
4	1	5	4	0
	−1	0	0	
4	4	5	6	7
	0	0	1	

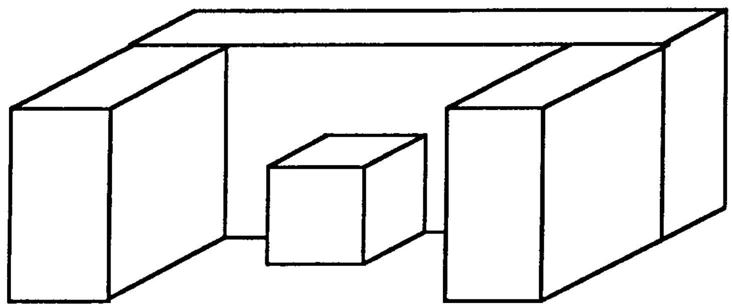
8	6			
0	0	.5		
7.5	1.5	1		
−1	−1	−2		
−1	1	−2		
1	1	−2		
1	−1	−2		
−1	−1	2		
−1	1	2		
1	1	2		
1	−1	2		
4	0	1	2	3
	0	0	−1	
4	0	4	7	3
	0	−1	0	
4	3	7	6	2
	1	0	0	
4	2	6	5	1
	0	1	0	
4	1	5	4	0
	−1	0	0	
4	4	5	6	7
	0	0	1	

```
3
0 1 2
3

0
5.5 1.5 3
0 0 -1
-1 0 0
0 -1 0
1.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5

0
5.5 1.5 3
0 0 -1
-1 0 0
0 -1 0
2.5
0

0
0 -10 0
0 1 0
0 0 1
1 0 0
20
0
```

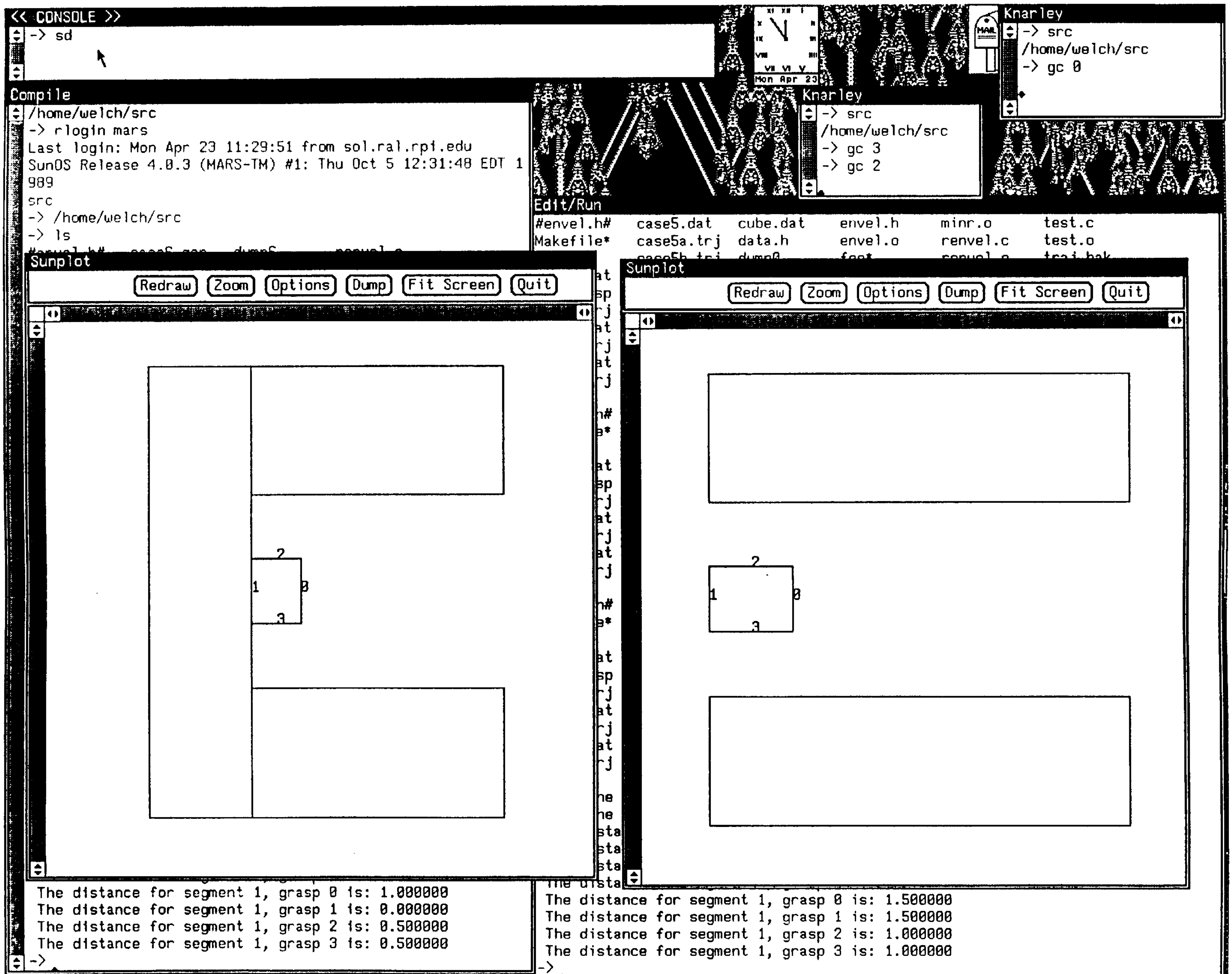
Case 5: Test of Assembly Order

This test case is used to demonstrate the plane sweep algorithm's ability to choose more appropriate assembly orders.

Associated Files: ~/case5.dat
~/case5a.trj
~/case5b.trj



Case 5. Example test case for ranking assembly operation orderings. A small obstacle in a semi-cluttered environment with a nearby sidewall.



<< CONSOLE >>
-> sd
Compile
/home/welch/src
-> rlogin mars
Last login: Mon Apr 23 11:29:51 from sol.ral.rpi.edu
SunOS Release 4.0.3 (MARS-TM) #1: Thu Oct 5 12:31:48 EDT 1
989
src
-> /home/welch/src
-> ls
Knarley
-> src
/home/welch/src
-> gc 0
Knarley
-> src
/home/welch/src
-> gc 3
-> gc 2
Edit/Run
#envel.h# case5.dat cube.dat envel.h minr.o test.c
Makefile* case5a.trj data.h envel.o renvel.c test.o
Sunplot
Redraw Zoom Options Dump Fit Screen Quit
2
1
0
3
Sunplot
Redraw Zoom Options Dump Fit Screen Quit
2
1
0
3
The distance for segment 1, grasp 0 is: 1.000000
The distance for segment 1, grasp 1 is: 0.000000
The distance for segment 1, grasp 2 is: 0.500000
The distance for segment 1, grasp 3 is: 0.500000
The distance for segment 1, grasp 0 is: 1.500000
The distance for segment 1, grasp 1 is: 1.500000
The distance for segment 1, grasp 2 is: 1.000000
The distance for segment 1, grasp 3 is: 1.000000

```
4

8       6
0       0       1
0       0       0
0       0       0
0       2       0
5       2       0
5       0       0
0       0       2
0       2       2
5       2       2
5       0       2
4       0       1       2       3
        0       0       -1
4       0       4       7       3
        0       -1      0
4       3       7       6       2
        1       0       0
4       2       6       5       1
        0       1       0
4       1       5       4       0
        -1      0       0
4       4       5       6       7
        0       0       1

8       6
0       0       1
0       5       0
0       0       0
0       2       0
5       2       0
5       0       0
0       0       2
0       2       2
5       2       2
5       0       2
4       0       1       2       3
        0       0       -1
4       0       4       7       3
        0       -1      0
4       3       7       6       2
        1       0       0
4       2       6       5       1
        0       1       0
4       1       5       4       0
        -1      0       0
4       4       5       6       7
        0       0       1

8       6
0       0       1
4       3       0
0       0       0
0       1       0
1       1       0
1       0       0
0       0       1
0       1       1
1       1       1
1       0       1
4       0       1       2       3
        0       0       -1
4       0       4       7       3
```

```
        0       -1      0
4       3       7       6       2
        1       0       0
4       2       6       5       1
        0       1       0
4       1       5       4       0
        -1      0       0
4       4       5       6       7
        0       0       1

8       6
0       0       1
5       0       0
0       0       0
0       7       0
2       7       0
2       0       0
0       0       2
0       7       2
2       7       2
2       0       2
4       0       1       2       3
        0       0       -1
4       0       4       7       3
        0       -1      0
4       3       7       6       2
        1       0       0
4       2       6       5       1
        0       1       0
4       1       5       4       0
        -1      0       0
4       4       5       6       7
        0       0       1
```

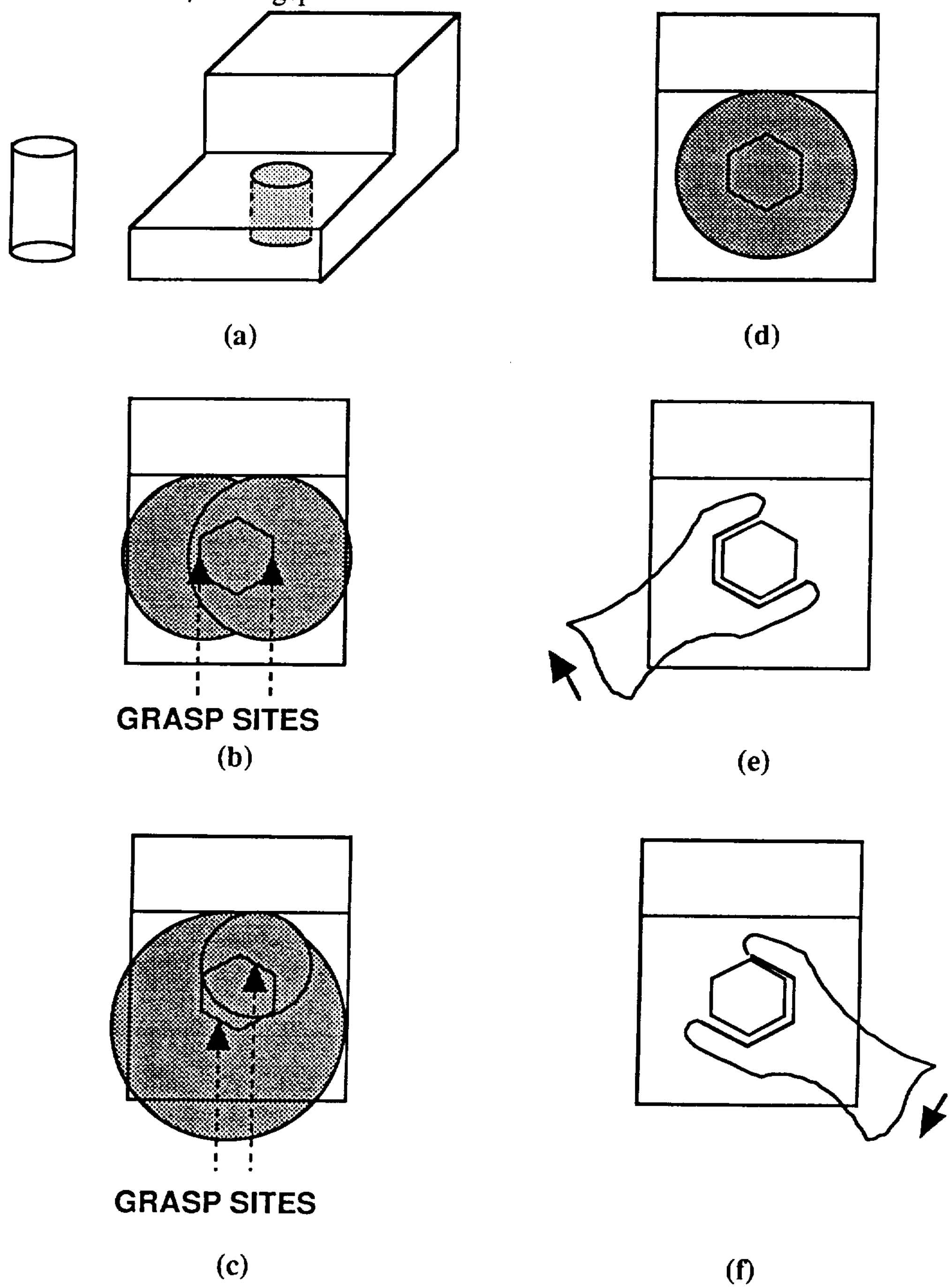
```
4
0 1 2 3
2

0
4.5 3.5 4
0 0 -1
-1 0 0
0 -1 0
3.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5

0
4.5 3.5 4
0 0 -1
-1 0 0
0 -1 0
2.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5
```

```
3
0 1 2
2

0
4.5 3.5 4
0 0 -1
-1 0 0
0 -1 0
3.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5

0
4.5 3.5 4
0 0 -1
-1 0 0
0 -1 0
2.5
4
0 .5 0
0 -.5 0
0 0 .5
0 0 -.5
```

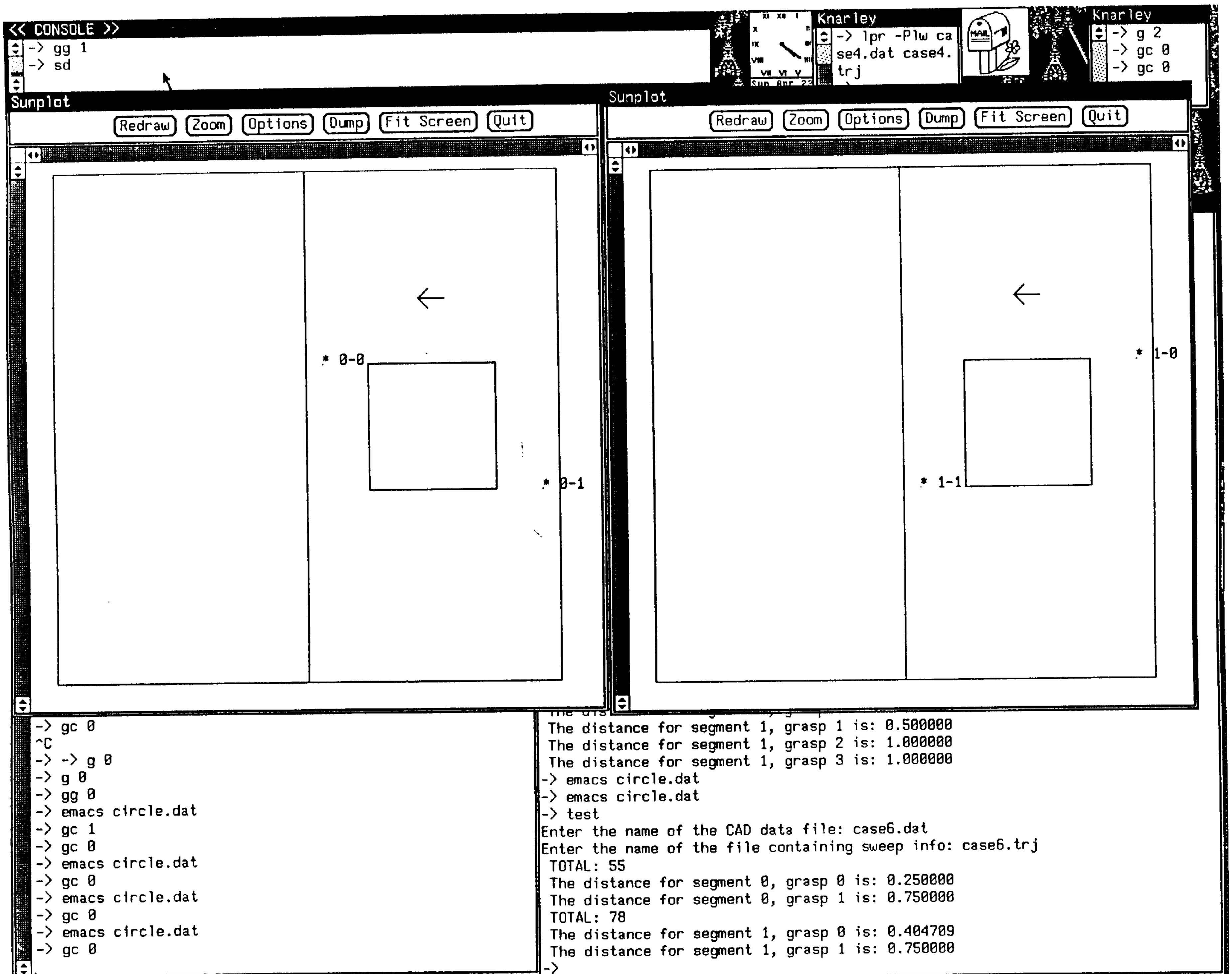
Case 6: Rotating Bolt Head

This test case is used to demonstrate the plane sweep algorithm's ability to choose more appropriate rotational grasp locations.

Associated Files: ~/case6.dat
~/case6.trj
~/case6.gsp



Case 6. (a) Peg in hole with side wall obstacle assembly. (b) and (c) The gripper envelope for a hex head bolt in place of the peg. (d) The gripper envelope for a socket attachment. (e) and (f) The effects of wrench placement which is detectable with a rotational sweep.



<< CONSOLE >>
-> gg 1
-> sd
Knarley
-> lpr -Plw ca
se4.dat case4.
trj
Knarley
-> g 2
-> gc 0
-> gc 0
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
* 0-0
* 0-1
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
* 1-0
* 1-1
-> gc 0
^C
-> -> g 0
-> g 0
-> gg 0
-> emacs circle.dat
-> gc 1
-> gc 0
-> emacs circle.dat
-> gc 0
-> emacs circle.dat
-> gc 0
-> emacs circle.dat
-> gc 0
The distance for segment 1, grasp 1 is: 0.500000
The distance for segment 1, grasp 2 is: 1.000000
The distance for segment 1, grasp 3 is: 1.000000
-> emacs circle.dat
-> emacs circle.dat
-> test
Enter the name of the CAD data file: case6.dat
Enter the name of the file containing sweep info: case6.trj
TOTAL: 55
The distance for segment 0, grasp 0 is: 0.250000
The distance for segment 0, grasp 1 is: 0.750000
TOTAL: 78
The distance for segment 1, grasp 0 is: 0.404709
The distance for segment 1, grasp 1 is: 0.750000
->

```
2
20      12
0       0       1
0       0       0
0       2       0
0       2       1
2       2       1
2       2       3
4       2       3
4       2       0
0       -2      0
0       -2      1
2       -2      1
2       -2      3
4       -2      3
4       -2      0
1.5     -.5     0
1.5     .5      0
1.5     -.5     1
1.5     .5      1
4       0       6       11      5
        0       0       -1
6       0       1       2       3       4       5
        0       1       0
6       6       7       8       9       10      11
        0       -1      0
4       0       6       7       1
        -1      0       0
4       1       7       8       2
        0       0       1
4       2       8       9       3
        -1      0       0
4       3       9       10      4
        0       0       1
4       4       10      11      5
        1       0       0
4       12      16      17      13
        1       0       0
4       13      17      18      14
        0       -1      0
4       14      18      19      15
        -1      0       0
4       15      19      16      12
        0       1       0

8       6
0       0       .5
1       0       1
-1      -1      -2
-1      1       -2
1       1       -2
1       -1      -2
-1      -1      2
-1      1       2
1       1       2
1       -1      2
4       0       1       2       3
        0       0       -1
4       0       4       7       3
        0       -1      0
4       3       7       6       2
        1       0       0
4       2       6       5       1
        0       1       0
```

4	1	5	4	0
	−1	0	0	
4	4	5	6	7
	0	0	1	

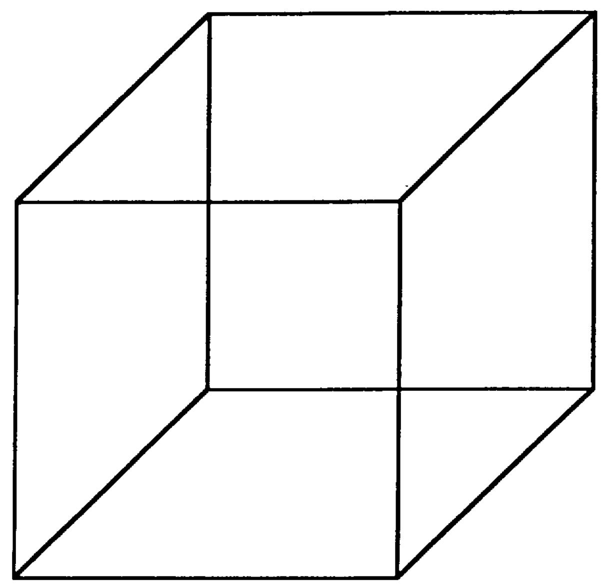
```
-.866 -.5 -.866 -.5 "* 1-1"
-.1 1 -.025 1.075 " "'-F
```

```
1
0
2

1
1 0 1.75
0 0 1
1.866 -.5 1.75
1.6
2
0 .75 0
0 -.75 0

1
1 0 1.75
0 0 1
1.866 .5 1.75
1.6
2
0 .75 0
0 -.75 0
```

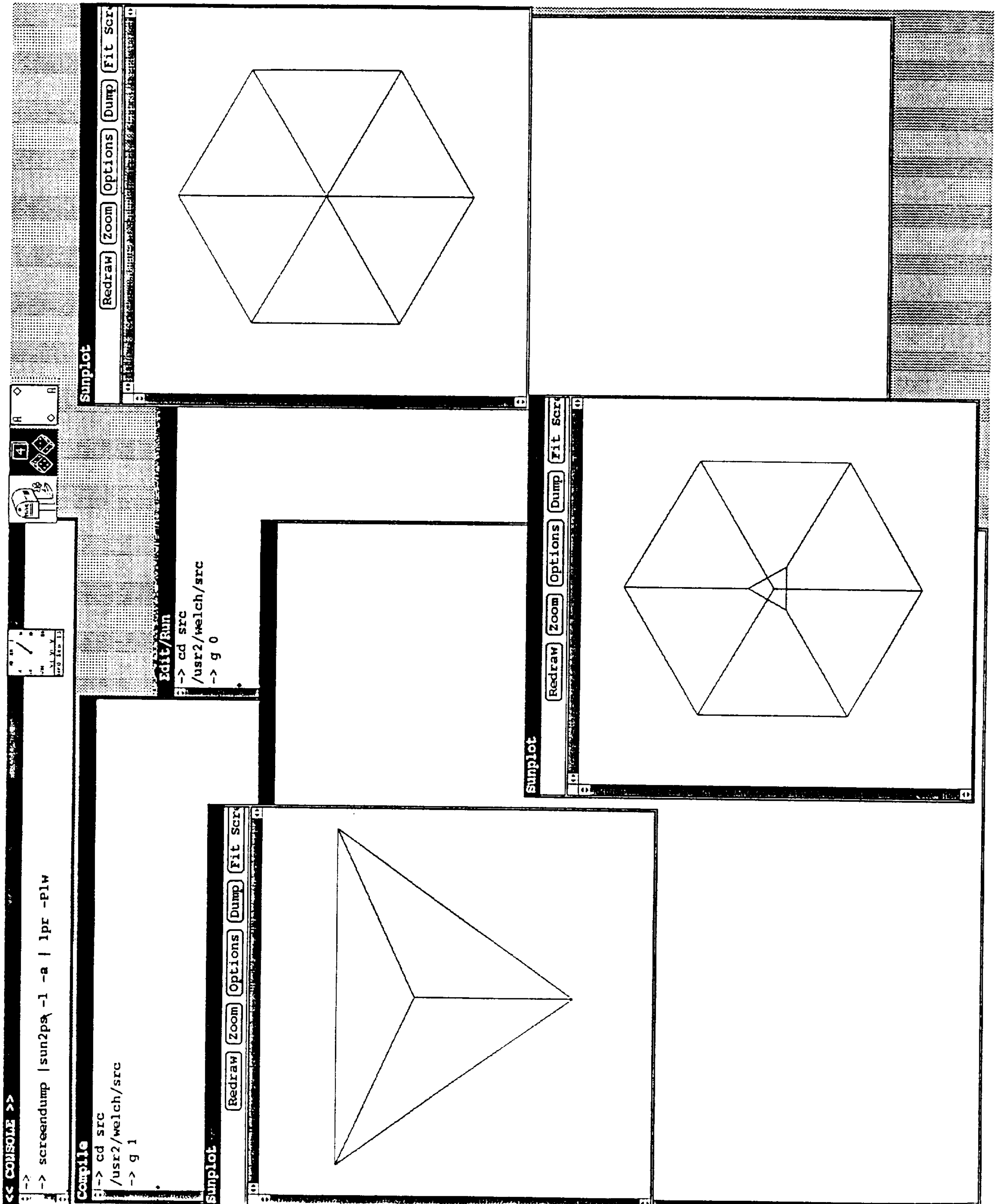
Sweep Shadows of a Cube

This test case is used to demonstrate the various kinds and types of sweep shadows that might be generated by the plane sweep algorithm.

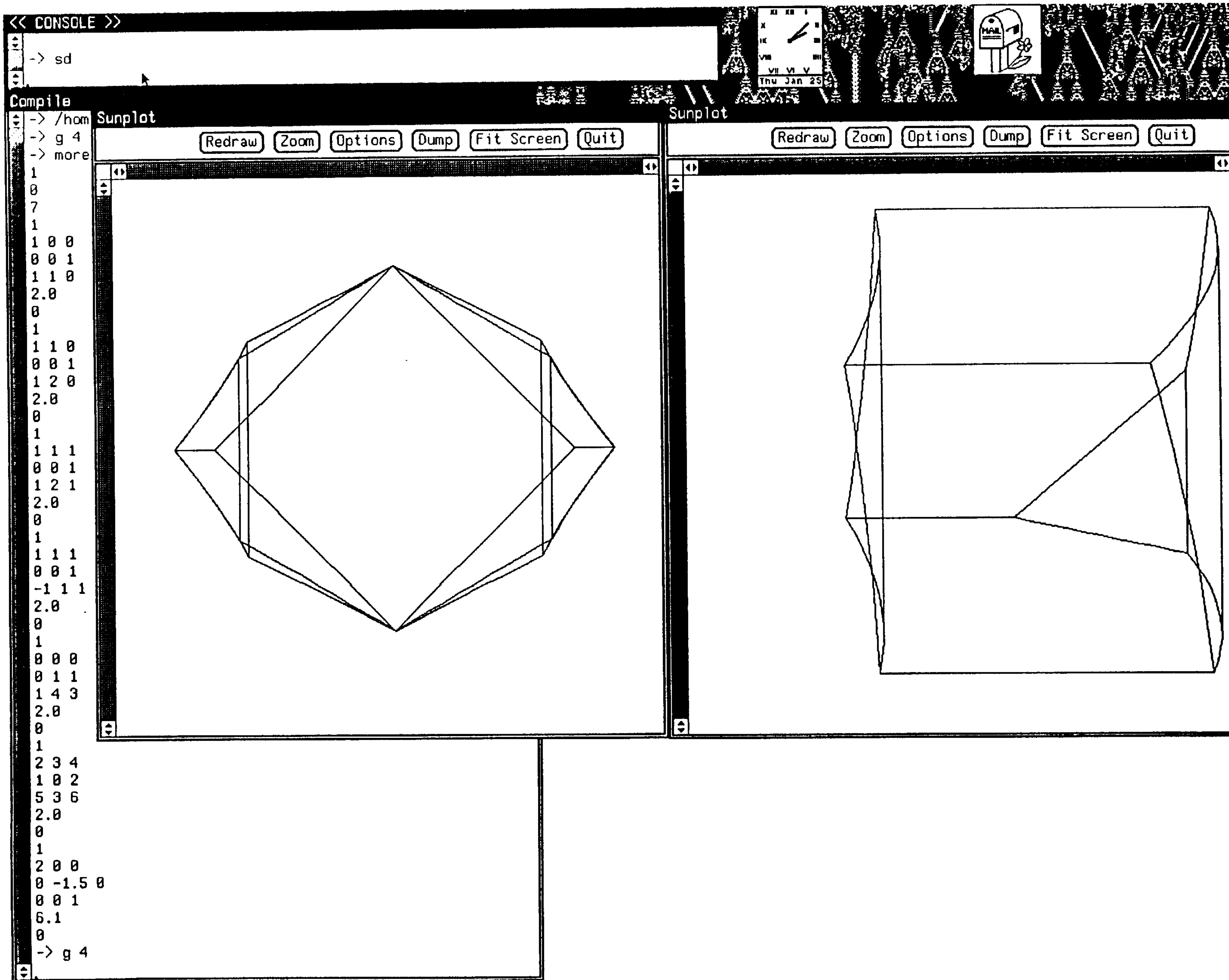
Associated Files: ~/cube.dat
~/cube.trj



Cube. The wire-frame of a cube used to test sweep shadow generation.



PRECEDING PAGE BLANK NOT FILMED



<< CONSOLE >>
-> sd
Compile
-> /hom
-> g 4
-> more
1
0
7
1
1 0 0
0 0 1
1 1 0
2.0
0
1
1 1 0
0 0 1
1 2 0
2.0
0
1
1 1 1
0 0 1
1 2 1
2.0
0
1
1 1 1
0 0 1
-1 1 1
2.0
0
1
0 0 0
0 1 1
1 4 3
2.0
0
1
2 3 4
1 0 2
5 3 6
2.0
0
1
2 0 0
0 -1.5 0
0 0 1
6.1
0
-> g 4
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
Sunplot
Redraw
Zoom
Options
Dump
Fit Screen
Quit
Thu Jan 25

```
1
8       6       0.0     0.0     4.0     0.0     0.0     0.0
-0.25   -0.25   -0.25
-0.25   -0.25    0.25
-0.25    0.25   -0.25
-0.25    0.25    0.25
 0.25   -0.25   -0.25
 0.25   -0.25    0.25
 0.25    0.25   -0.25
 0.25    0.25    0.25
4
0       1       3       2
-1.0     0.0     0.0
4
4       5       7       6
 1.0     0.0     0.0
4
0       2       6       4
 0.0     0.0    -1.0
4
1       3       7       5
 0.0     0.0     1.0
4
0       1       5       4
 0.0    -1.0     0.0
4
2       3       7       6
 0.0     1.0    0.0'-F
```

```
1
0
10

0
-2        -2        -2
2.5
0

0
-2        -2        -2
4.5
0

0
-2        -2        -2
6
0

1
1 0 0
0 0 1
1 1 0
2.0
0

1
1 1 0
0 0 1
1 2 0
2.0
0

1
1 1 1
0 0 1
1 2 1
2.0
0

1
1 1 1
0 0 1
-1 1 1
2.0
0

1
0 0 0
0 1 1
1 4 3
2.0
0

1
2 3 4
1 0 2
5 3 6
2.0
0

1
2 0 0
0 -1.5 0
```

```
0 0 1
6.1
0
```

Miscellaneous

This test case is used to generate the orthonormal views of most reasonably sized wire-frame objects.

Associated Files: ~/view.trj

```
4
0 1 2 3
3

0
-10 0 0
1 0 0
0 1 0
0 0 1
20
0

0
0 -10 0
0 1 0
0 0 1
1 0 0
20
0

0
0 0 -10
0 0 1
1 0 0
0 1 0
20
0
```

Appendix B
OnLine User's Document

Gripper Envelope Detection Using Sweep Shadows

Henry L. Welch

March 1992

Abstract

The gripper envelope analysis software derives a single metric for approximating the volume available for a robot's end-effector during mating operations. The basic approach of the software is to generate the straight line and rotational trajectory sweep shadows of obstacles in the environment and to find the closest distance from grasp sites to these shadows. The obstacles are limited to objects composed of planar faces described using a surface modeling technique.

1 Introduction

The size of the gripper envelope is important in the domain of assembly sequence planning. Its role is even more important if this planning is undertaken without a priori knowledge of the size and shape of the robot performing the assembly. The distance metric supplied by this code is intended as a single value which characterizes the size of the gripper envelope.

The underlying assumption in this solution to gripper envelope detection is that during a motion there is no relative motion between the object being grasped and the robot's gripper. This means that for certain classes of motion the cross-sectional area swept by the object to be mated (and hence the robot's gripper) is constant along the entire length of the trajectory. The two simplest examples of these trajectories are straight line segments and simple rotations.

Since the cross-sectional area is constant, the cross-sectional shape of the gripper envelope can be found by sweeping a plane along the motions of the object. As the plane encounters environmental obstacles, areas in the plane are swept out which represent the sweep shadows of these obstacles. For the cases of straight lines segments and simple rotations this is very similar to projection in cartesian and cylindrical coordinates respectively. To account for obstacles which may only be partially swept by the plane, clipping and closure are also required.

Sweep parameters are obtained by converting each sweep trajectory to a local cartesian coordinate system about the trajectory. Obstacle data is then converted to this coordinate system and then processed appropriately.

The reported size of the gripper envelope then depends on the location of the grasp site within the sweep plane. By inscibing the largest possible circle centered about each grasp site, a single value can be used to describe the approximate size of the gripper envelope. This value is calculated by determining the closest sweep shadow point to the grasp site.

The object modeling used by this software is to describe an object as a set of faces. Each face is then defined as a directed list of vertices (which describe the edges of the face) with closure between the last and first vertices assumed. Normals for each surface are also required, but they are currently not required to be directed outward.

NOTE: This software requires that all trajectories and grasp sites supplied by the user do not intersect with any of the obstacles.

For more information on these algorithms and possible applications of the gripper envelope see "Solution to the Gripper Envelope Problem Using a Planar Sweep," CIRSSE Report 111, Rensselaer Polytechnic Institute and "Robot Independent Assembly Sequence Planning," Ph.D. Thesis, Rensselaer Polytechnic Institute, August 1990.

2 Requirements

The gripper envelope code was developed on a SUN3/60 workstation under the Unix 4.0.2 operating system. It must be linked to the standard I/O library and the math library. This code was recompiled and tested on a SUN4 system with no difficulties.

In order to use the aliases in Section 4.2, the graph and sunplot utilities must be available and supported on your system. If they are not available, an utility which plots unconnected line segments followed by labels in double quotes will perform the same function.

Very little consistency checking on the input data is performed. This requires that the user makes sure that CAD objects are consistent and that all cartesian coordinate systems are defined by orthogonal bases.

It is very important to remember that the gripper envelope code assumes that all trajectories are collision-free and that grasp sites do not intersect the obstacles. Correct results are not guaranteed when these connections are not met.

3 Build Procedure

Installation of the gripper envelope software is facilitated by the inclusion of a Makefile. Currently the Makefile compiles and links the gripper envelope code with the file test.c. This can be easily updated for your particular application by modifying the entries for test and all in the Makefile.

To compile and link the software, simply execute the Unix command 'make all' from the directory containing all the gripper envelope code.

4 User's Guide

The primary usage of the gripper envelope software is to generate and test the sweep shadows of environments. The basic flow of the software is:

1) Read CAD data

2) Preprocess CAD data to internal form

3) Load Trajectory data

4) Sweep the CAD data through the trajectory

5) Load Grasp data

6) Test each Grasp Point

7) Save necessary information

Code performing this basic function is provided in the file test.c. The program test will prompt the user for two file specifications with the default directory being the current one. The first specification is for the CAD data file. The second contains the trajectory and grasp information to test.

The following deficiencies in the algorithm are currently known.

1) The dump_rot_plane routine does not always generate line segments which completely describe the curves they are to represent. This may result in object edges that do not always meet at vertices. In certain extreme cases, this may result in slightly erroneous results from the distance generation function.

2) Planar faces with holes are not currently supported by the CAD data structure. To work around this defficiency, add an extra edge which connects the hole to the outer edge of the face and divide the face into two faces. This will generate extra edges in the sweep plane, but will not effect the results of the distance geneation functions.

4.1 Data File Formats

There are two types of input file used by the test feature of the gripper envelope system. The first is used by the read_CAD routine and processes a surface model of all the objects in the current system. The format of this file is:

```
Number_of_Objects

/* For each object */
Number_of_Vertices   Number_of_Faces

/* Orientation and positioning data w.r.t. default origin */
Theta_Rotation       Phi_Rotation         Scale
X_Translation        Y_Translation        Z_Translation

/* For each vertex */
Vertex_X             Vertex_Y             Vertex_Z

/* For each face */
Vertex_Count         Vertex_List

Normal_X             Normal_Y             Normal_Z
```

Many examples of the use of this data style can be found in the case#.dat files. To view the orthonormal projections of objects along the x-, y-, and z-axes run the test code with your CAD data file and the view.trj trajectory file and then plot the dump files. Be sure to modify the top of view.trj so the proper number of objects are specified.

The second type of input data file for the test system contains the trajectory and grasp information. The format for this file is:

```
Number_of_Objects_Assembled /* This may differ from the CAD file */
```

```
List_of_Assembled_Objects
Number_of_Trajectories

/* For each Trajectory */
Trajectory_Type

/* If Trajectory_Type == 0 then straight line trajectory */
Origin_X             Origin_Y             Origin_Z

/* The next three vectors describe a unit orthogonal basis */
Trajectory_Dir_X     Trajectory_Dir_Y     Trajectory_Dir_Z
Basis2_X             Basis2_Y             Basis2_Z
Basis3_X             Basis3_Y             Basis3_Z

Length_of_Trajectory

/* Else if Trajectory_Type == 1 then rotational trajectory */
Origin_X             Origin_Y             Origin_Z
Rot_Normal_X         Rot_Normal_Y         Rot_Normal_Z
Reference_X          Reference_Y          Reference_Z

Angle_to_Rotate_Through

/* EndIf */

Number_of_Grasp_Sites

/* For each Grasp Site */
Grasp_Type
Grasp_X              Grasp_Y              /* In sweep coordinates */

/* If Grasp_Type == 1 then only half the sweep need be checked */
Normal_X             Normal_Y
```

4.2 Useful Aliases

Some useful aliases when using the dump_plane and dump_rot_plane routines to display the data.

alias g graph -g 0 -b <dump\!* | sunplot

alias gg graph -b <dump\!* | sunplot

alias gc cat dump\!* grasp.dat | graph -g 0 -b | sunplot

These aliases use graph and sunplot to generate visual representations of the sweep data without a grid, with a grid, and with grasp data respectively. They are to be used with a single parameter which is a number representing the trajectory number to display. These correspond to the dump# files created by the dump plane routines.

A Manifest

This appendix contains a list of all the files included with the gripper envelope software package.

A.1 Initialization Files

Makefile – The make file for the source code. Invoked 'make all'.

const.h – Constants used throughout the code.

data.h – CAD data structure used.

envel.h – Special straight line data structures.

renvel.h – Special rotational data structures.

info.tex – Generates this report.

README – Tells how to generate this report.

A.2 Code Files

envel.c – Support routines for straight line sweeps.

minr.c – Computes the minimal radius line for a face.

renvel.c – Support routines for rotational sweeps.

rotate.c – Computes the rotational sweeps.

sweep.c – Conputes the straight line sweeps.

sweepio.c – I/O routines for the package.

A.3 Test Case Files

test.c – Sample calling frame for using the gripper envelope code.

case1.dat
case1.trj – A straight line test case of a two D-cell flashlight.
Tests GRASP TYPE

case2.dat
case2.trj
case2.gsp – A straight line test case of a peg-in-hole with nearby obstructing wall.
Tests GRASP LOCATION

case3.dat
case3.trj – A straight line test case of a peg in a pinched box.
Tests GRASP LOCATION

case4.dat
case4.trj – A straight line test case of a line of indicators.
Tests GRASP LOCATION

case5.dat
case5a.trj
case5b.trj – A straight line test case of the effects of assembly order.
Tests ASSEMBLY ORDER/TRAJECTORIES

case6.dat
case6.gsp
case6.trj – A rotational test case similar to case 2, but with the rotation of a bolt head.
Tests ROTATIONAL GRASP LOCATION

cube.dat
cube.trj – The original test object with multiple straight line and rotational trajectories. No grasp sites.
Tests SWEEP CALCULATIONS

grasp.dat – The location of grasp sites in the test case.

view.trj – Orthonormal views of the objects along the primary axes.

B Error Messages

There is very little data checking that takes place within the gripper envelope software. This is due primarily to its role as a subfunctin within an assembly sequence planner which generates consistent coordinate data.

The routine read_CAD reports when the CAD file input is not found and prompts the user for another.

The test code also reports an inappropriate trajectory file name and prompts the user for another.

Appendix C
Software Listings

```
# /************************************************************************ /
# /*                                                                      */
# /* This Makefile provides the necessary commands to compile and install */
# /* the planar sweep alogorithms test code.                              */
# /*                                                                      */
# /* Written by: HLW June 1990                                            */
# /*                                                                      */
# /************************************************************************ /

# Host C-compiler
CFLAGS = -O -fsingle
LIBS = -lm

# Host object files
COBJ = sweep.o envel.o test.o sweepio.o renvel.o rotate.o minr.o
TNGON =

all: test

test: $(COBJ)
		$(CC) $(CFLAGS) -o test $(COBJ) $(LIBS)

test.o: test.c envel.h data.h const.h

minr.o: minr.c envel.h data.h const.h

sweep.o: sweep.c envel.h data.h const.h

envel.o: envel.c envel.h const.h

sweepio.o: sweepio.c envel.h data.h const.h

renvel.o: renvel.c envel.h const.h

rotate.o: rotate.c envel.h data.h const.h

clean:
		rm -f *.o *.obj *.asm *.map *_map.h *.lst *% core ngon.abs

install: test
		mv test /home/welch/bin

spotless: clean
		sccs clean
```

```
/***********************************************************************
 *  envel.c  — This module contains the support routines to handle    *
 *              the 2-D plane sweep for the gripper envelope problem. *
 *                                                                     *
 *  Written by: HLW June, 1989                                         *
 ***********************************************************************/
/* The following conventions are assumed throughout:

    1) Coordinates are seen as row matrices.
    2) Transformation matrices are right-multiplied.
*/

#include <stdio.h>
#include <math.h>
#include "envel.h"
#include "const.h"

void mat3inv (inmat, outmat)
float   inmat[3][3];                            /* Input matrix */
float   outmat[3][3];                           /* Inverse matrix */

/* The routine mat3inv, inverts the matrix inmat using Gauss-Jordan
   elimination and returns the result in the matrix outmat.
   The matrices are all considered to be 3x3. */
/* This routine ASSUMES that the input matrix is non-singular. */

{
#define size 3                                  /* The matrix sizes */
    int     i,j,k;                              /* Loop indices */
```

```
    float pivot;                                        /* The pivot value */
    float null;                                         /* The value to null */
    float swap;                                         /* Temp. swap storage */
    float tmpmat[3][3];                                 /* Scratch storage */

    /* Initialize the output matrix to the identity */
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            tmpmat[i][j] = inmat[i][j];
            outmat[i][j] = 0.0;
        }
        outmat [i][i] = 1.0;
    }
    /* Begin scanning the rows */
    for (i = 0; i < size; i++) {
        /* check the pivot element */
        pivot = tmpmat[i][i];
        if (pivot == 0.0) {
            /* scan for a non-zero pivot element */
            j = i + 1;
            while (tmpmat[j][i] == 0.0) j++;
            /* swap the rows */
            for (k = 0; k < size; k++) {
                swap = tmpmat[i][k];
                tmpmat[i][k] = tmpmat[j][k];
                tmpmat[j][k] = swap;
                swap = outmat[i][k];
                outmat[i][k] = outmat[j][k];
                outmat[j][k] = swap;
            }
            pivot = tmpmat[i][i];
        } /* end if (pivot */

        /* invert the pivot and normalize the diagonal element */
        pivot = 1.0 / pivot;
        for (k = 0; k < size; k++) {
            tmpmat[i][k] = tmpmat[i][k] * pivot;
            outmat[i][k] = outmat[i][k] * pivot;
        }

        /* clear the pivot element's column */
        for (j = 0; j < size; j++)
            if (j != i) {
                null = -tmpmat[j][i];
                for (k = 0; k < size; k++) {
                    tmpmat[j][k] += null * tmpmat[i][k];
                    outmat[j][k] += null * outmat[i][k];
                }
            } /* end if (j != */
    } /* end for (i = 0 */
} /* end routine mat3inv */

void mat4mul (mat1, mat2, outmat)
float mat1[4][4], mat2[4][4], outmat[4][4];        /* The 4x4 matrices */

/* The routine mat4mul multiplies the two 4x4 matrices mat1 and mat2 to
   find outmat. Mat1 will be multiplied left of mat2. */

{
    int i,j,k;                                          /* Loop indices */
#define sz 4
```

```
    /* loop on rows */
    for (i = 0; i < sz; i++)
        /* loop on columns */
        for (j = 0; j < sz; j++) {
            outmat[i][j] = 0.0;
            /* perform summing multiplication */
            for (k = 0; k < sz; k++)
                outmat[i][j] += mat1[i][k] * mat2[k][j];
        }
  } /* routine mat4mul */

void do_xfm (xfm, invert, outvert)
float  xfm[4][4];                                    /* Transformation matrix */
struct t_coord4 invert;                              /* Incoming vertex */
struct coord3 *outvert;                              /* Transformed vertex */

/* The routine do_xfm performs the homogeneous transformation xfm to the
   vertex invert and generates a 3-D point outvert */

{
    int     i;                                       /* Loop indices */

    /* loop on each coordinate of the vertex */
    for (i = 0; i < 3; i++) {
        outvert->pt[i] = 0.0;
        /* perform coordinate transformation */
        outvert->pt[i]  = invert.x * xfm[0][i];
        outvert->pt[i] += invert.y * xfm[1][i];
        outvert->pt[i] += invert.z * xfm[2][i];
        outvert->pt[i] += invert.w * xfm[3][i];
    } /* for (i */
  } /* routine do_xfm */


void findloc (point, origin, basis, loc)
float point[3];                                      /* Point to project */
float origin [3];                                    /* Origin for basis */
float basis[3][3];                                   /* Inverted basis matrix */
float loc[3];                                        /* Location in basis space */

/* The routine findloc projects a point onto a basis w.r.t an arbitrary
   coordinate system origin.  The result is returned in loc. */

{
    int     i,j;                                     /* Loop indices */
    float   diff[3];                                 /* Difference vector */

    /* find the location of the point relative to the origin */
    for (i = 0; i < 3; i++)
        diff[i] = point[i] - origin[i];

    /* find the location via matrix multiplication */
    for (i = 0; i < 3; i++) {
        loc[i] = 0.0;
        for (j = 0; j < 3; j++)
            loc[i] += basis[j][i] * diff[j];
    } /* for (i = 0 */
} /* routine findloc */

int sweepseg (origin, basis, length, p1, p2, segment)
float origin[3];                                     /* Origin for basis */
```

```
float basis [3][3];                              /* Inverted basis matrix */
float length;                                    /* Length of sweep */
float p1[3],p2[3];                               /* Endpoints of segment */
struct seg *segment;                             /* 2-D segment */

/* The routine sweepseg sweeps a plane described by an origin and basis
   along the trajectory traj for a distance length.  The endpoints of the
   line segment defined by p1 and p2 in the swept plane are returned as
   2-D coordinates in op1 and op2.  If the segment is not swept by the plane,
   then the routine returns 0; otherwise it returns 1. */

{
    float proj1[3],proj2[3];                     /* The projections of p1, p2 */
    float z1, z2;                                /* Traj. direction coords. */
    float diff[3];                               /* Projection differences */
    int   i;                                     /* A loop index */

    /* project the two points */
    findloc (p1, origin, basis, proj1);
    z1 = proj1[0];
    findloc (p2, origin, basis, proj2);
    z2 = proj2[0];

    for (i = 0; i < 3; i++)
        diff[i] = proj2[i] - proj1[i];
    /* Determine if the segment is even swept */
    if ((z1 < 0.0) & (z2 < 0.0))
        return(0);
    else if ((z1 > length) & (z2 > length))
        return(0);
    else {
        /* clip the segment if it needs it */
        /* deal with point 1 first */
        if (z1 < 0.0) {
            proj1[1] += -z1 * diff[1] /diff[0];
            proj1[2] += -z1 * diff[2] /diff[0];
        }
        else if (z1 > length) {
            proj1[1] += (length - z1) * diff[1] /diff[0];
            proj1[2] += (length - z1) * diff[2] /diff[0];
        }
        /* deal with point 2 */
        if (z2 < 0.0) {
            proj2[1] += -z2 * diff[1] /diff[0];
            proj2[2] += -z2 * diff[2] /diff[0];
        }
        else if (z2 > length) {
            proj2[1] += (length - z2) * diff[1] /diff[0];
            proj2[2] += (length - z2) * diff[2] /diff[0];
        }
        /* clipping complete, copy the results and return */
        for (i = 0; i < 2; i++) {
            segment->p1[i] = proj1[i+1];
            segment->p2[i] = proj2[i+1];
        }
        return(1);
    } /* else */
} /* routine sweepseg */

void d_pt2seg(point, segment, d, pc)
float point[2];                                  /* point in 2D */
```

```
struct seg segment;                                        /* segment end-points */
float *d;                                                  /* the closest distance */
float pc[2];                                               /* the closest point */

/* The routine d_pt2seg determines the closest point on the line segment
   p1 p2 to point and returns that point as pc, along with the distance d^2.
   The routine generates a line from p1 toward p2 and determines the closest
   point on this line as represented by the parameter m.  If m is not in
   the range [0,1] then the appropriate endpoint is returned. */

{
   float m;                                                /* projection on segment */
#define x1 segment.p1[0]
#define y1 segment.p1[1]
#define x2 segment.p2[0]
#define y2 segment.p2[1]
#define x  point[0]
#define y  point[1]

    /* find the value of m that satisfies the equation that the first
       derivative of the euclidean distance is 0. */
   m = ((x2 - x1)*(x2 - x1) + (y2 - y1)*(y2 - y1));

   if (m == 0) {  /* The segment is only a point */
      pc[0] = x1;
      pc[1] = y1;
    }
   else {  /* process normally */
      m = ((x2 - x1)*(x  - x1) + (y2 - y1)*(y  - y1))  / m;

       /* clip m */
      if (m > 1.0) m = 1.0;
      if (m < 0.0) m = 0.0;

       /* find the distance and the point to return */
      pc[0] = x1 + m*(x2 - x1);
      pc[1] = y1 + m*(y2 - y1);
    }  /* end else of if (m == 0 */

   *d = (x - pc[0])*(x - pc[0]) + (y - pc[1])*(y - pc[1]);
 }  /* routine d_pt2seg */

int half_plane(inplane, incount, origin, normal, outplane, outcount)
struct seg inplane[2*TOT_VERTS];                         /* Unclipped segments */
int      incount;                                        /* # of segments */
float    origin[2];                                      /* point on clipping plane */
float    normal[2];                                      /* outward normal of plane */
struct seg outplane[2*TOT_VERTS];                        /* Clipped half plane */
int      *outcount;                                      /* # of output vertices */

/* The routine half_plane clips a set of line segments within a plane so
   that they lie in the half plane defined by a normal and point.  The
   resulting segments and their number are computed with the count being
   returned.

   NOTE:  This routine assumes no overlap of the segments which define the
          swept objects and the origin point of the clipping plane. */

{
   int    i,j;                                           /* Loop indices */
   float dot1, dot2;                                     /* Dot products */
```

```
float proj;                                          /* Point locator */

/* initialize the output count and loop on all the input segments */
*outcount = 0;
for (i = 0; i < incount; i++) {

    /* Find the dot products of the endpoints with the normal */
    dot1 = dot2 = 0.0;
    for (j = 0; j < 2; j++) {
        dot1 += normal[j] * (inplane[i].p1[j] - origin[j]);
        dot2 += normal[j] * (inplane[i].p2[j] - origin[j]);
    }

    /* Check for clipping of the endpoints */
    if ((dot1 >= 0.0) & (dot2 >= 0.0)) {
        for (j = 0; j < 2; j++) {
            outplane[*outcount].p1[j] = inplane[i].p1[j];
            outplane[*outcount].p2[j] = inplane[i].p2[j];
        }
        *outcount = *outcount + 1;
    }
    else if (((dot1  < 0.0) & (dot2 >= 0.0)) |
            ((dot2 >= 0.0) & (dot2  < 0.0))) {
        /* One end must be clipped */
        proj = normal[0] * (inplane[i].p1[0] - origin[0]) +
                normal[1] * (inplane[i].p1[1] - origin[1]);
        proj = proj / (normal[0] * (inplane[i].p1[0] - inplane[i].p2[0]) +
                       normal[1] * (inplane[i].p1[1] - inplane[i].p2[1]));

        /* See which end */
        if (dot1 < 0.0)
            for (j = 0; j < 2; j++) {
                outplane[*outcount].p1[j] = inplane[i].p1[j] * (1.0 - proj)
                                          + inplane[i].p2[j] * proj;
                outplane[*outcount].p2[j] = inplane[i].p2[j];
            }
        else for (j = 0; j < 2; j ++) {
                outplane[*outcount].p2[j] = inplane[i].p1[j] * (1.0 - proj)
                                          + inplane[i].p2[j] * proj;
                outplane[*outcount].p1[j] = inplane[i].p1[j];
            }
        *outcount = *outcount + 1;
    } /* else if (((dot1 */
} /* for (i = 0 */
return(*outcount);
} /* routine half_plane */
```

```
/*
**                  NOTICE OF COPYRIGHT
**        Copyright (C) Rensselaer Polytechnic Institute.
**                1990 ALL RIGHTS RESERVED.
**
**
** Permission to use, distribute, and copy is granted ONLY
** for research purposes, provided that this notice is
** displayed and the author is acknowledged.
**
** This software is provided in the hope that it will be
** useful. BUT, in no event will the authors or Rensselaer
** be liable for any damages whatsoever, including any lost
** profits, lost monies, business interruption, or other
** special, incidental or consequential damages arising out
** of the use or inability to use (including but not
** limited to loss of data or data being rendered
** inaccurate or losses sustained by third parties or a
** failure of this software to operate) even if the user
** has been advised of the possibility of such damages, or
** for any claim by any other party.
**
** This software was developed at the facilities of the
** Center for Intelligent Robotic Systems for Space
** Exploration, Troy, New York, thanks to generous project
** funding by NASA.
**
** Package: Gripper Envelope Detection Using Sweep Shadows
**
** Written By: Henry L. Welch
**
*/
/**********************************************************************
 * minr.c — This module contains the support routines to handle        *
 *          the calculation of the minimal radius edge for             *
 *          rotational sweep shadows.                                  *
 *                                                                     *
 * Written by: HLW November, 1989                                      *
 **********************************************************************/
/* The following conventions are assumed throughout:

   1) Coordinates are seen as row matrices.
   2) Transformation matrices are right-multiplied.
*/

#include <stdio.h>
#include <math.h>
#include "envel.h"
#include "const.h"

int getminline(point, normal, refpt, step)

float point[3];                              /* A point on the plane */
float normal[3];                             /* Normal of the plane */
float refpt[3];                              /* Nominal start pt of line */
float step[3];                               /* Parametric step of line */

/* The routine getminline computes the minimal radius line on a plane for
   a rotational sweep shadow. It returns a value of 0 when the line
   exists and a value of 1 when the line degenerates to a point (ie
   z = const. */

{
```

```
#define A normal[0]                          /* Planar values in the */
#define B normal[1]                          /* form Ax + By + Cz = D */
#define C normal[2]
float D;
float temp;                                  /* Scratch storage */

    /* Compute the D parameter */
    D = A*point[0] + B*point[1] + C*point[2];
    /* Check for degenerate cases */
    if ((A == 0.0) & (B == 0.0))  /* Plane is constant z */
        return(1);
    else if (B == 0.0) {
        /* y defaults to zero */
        refpt[0] = D /A;
        refpt[1] = 0.0;
        refpt[2] = 0.0;
        step[0]  = -C /A;
        step[1]  = 0.0;
        step[2]  = 1.0;
        return(0);
    }
    else {  /* This is the normal case */
        temp = A*A + B*B;
        refpt[0] = A*D /temp;
        refpt[1] = (D - A*refpt[0]) /B;
        refpt[2] = 0.0;
        step[0]  = -A*C /temp;
        step[1]  = (D - C - A*(refpt[0] + step[0])) /B - refpt[1];
        step[2]  = 1.0;
        return(0);
    }
}  /* routine getminline */


int edge_clip(proj, refpt, step, loc)

struct tmp_rot *proj;                        /* Projected edge */
float   refpt[3];                            /* New edge ref. pt. */
float   step[3];                             /* Parametric step */
float   *loc;                                /* Parametric location */

/* The routine edge_clip determines the intersection point between the
   edge described in proj and the parametric line described by refpt and
   step.  The parametric location on the line is returned in loc.  Since
   there are many non-applicable possibilities, a return code is employed.
   Its values are: 0 - Intersection at a point
                   1 - Intersection at start point of edge
                   2 - Intersection at end point of edge
                   4 - Intersection along the entire edge
                   8 - No Intersection
*/

{
    float diff[3];                           /* Difference vector */
    int   i;                                 /* Loop index */
    float eloc;                              /* Location on the edge */
    int   parallel;                          /* Condition flag */
    int   same;

    /* Compute the difference vector for the edge */
    for (i = 0; i < 3; i++)
        diff[i] = proj->rp3[i] - proj->rp1[i];
```

```
    /* Check for edge and line parallel */
    parallel = TRUE;
    for (i = 0; i < 2; i++)
        if ((diff[2]*step[i]) != diff[i]) parallel = FALSE;

    /* Determine if the line is the same as the edge */
    if (parallel == TRUE) {
        same = TRUE;
        for (i = 0; i < 2; i++)
            if ((refpt[i] + proj->rp1[2]*step[i]) != proj->rp1[i]) same = FALSE;
        if (same == TRUE)
            return(4);
        else return(8);
    }
    else {  /* An intersection occurs */
        /* Compute the parametric intersection point on the edge */
        /* NOTE: step[2] = 1.0 and refpt[2] = 0.0 and are not included! */
        eloc = (diff[1] - step[1]*diff[2]);

        /* check for degenerate case */
        if (eloc == 0.0) {
            eloc = 1.0 /(diff[0] - step[0]*diff[2]);
            eloc = eloc * (refpt[0] - proj->rp1[0] + step[0]*proj->rp1[2]);

            /* if eloc is out of range, discontinue processing */
            if ((eloc < 0.0) | (eloc > 1.0))
                return(8);
            else {  /* Compute the intersection point */
                *loc = proj->rp1[2] + eloc*diff[2];
                /* return the appropriate code */
                if (eloc == 0.0) return(1);
                else if (eloc == 1.0) return(2);
                else return(0);
            }
        }
        else {
            eloc = 1.0 /eloc;
            eloc = eloc * (refpt[1] - proj->rp1[1] + step[1]*proj->rp1[2]);

            /* if eloc is out of range, discontinue processing */
            if ((eloc < 0.0) | (eloc > 1.0))
                return(8);
            else {  /* Compute the intersection point */
                *loc = proj->rp1[2] + eloc*diff[2];
                /* return the appropriate code */
                if (eloc == 0.0) return(1);
                else if (eloc == 1.0) return(2);
                else return(0);
            }
        }
    }  /* Intersection occurs */
}  /* Routine edge_clip */


int vcross(v1, v2, cross)

float v1[3];                                    /* Vector 1 */
float v2[3];                                    /* Vector 2 */
float cross[3];                                 /* Cross Product */
```

```
/* The routine vcross computes the cross product of two vectors and
   returns the result.  If v1 and v2 are parallel, then a return code
   of 1 is supplied; otherwise the return code is 0. */

{
    /* Compute the cross product */
   cross[0] = v1[1]*v2[2] - v1[2]*v2[1];
   cross[1] = v1[2]*v2[0] - v1[0]*v2[2];
   cross[2] = v1[0]*v2[1] - v1[1]*v2[0];

    /* Check for parallel vectors */
   if ((cross[0] == 0.0) & (cross[1] == 0.0) & (cross[2] == 0.0))
      return(1);
   else return(0);
 } /* Routine vcross */


int is_cross(v1, v2, ref)

struct tmp_rot *v1, *v2;                     /* Vectors to test */
float  ref[3];                                /* Reference vector */

/* The routine is_cross compares the cross products of the two edges with the
   reference vector to determine if a cross over occurs.  TRUE is returned
   when this occurs. */

{
   float d1[3],d2[3];                         /* Edge diff vectors */
   float m1,m2;                                  /* Magnitude values */
   float c1[3],c2[3];                         /* Cross products */
   int    dif;                                   /* Boolean flag */
   int    i;                                       /* Loop index */

    /* Generate the two edge difference vectors */
   for (i = 0; i < 3; i++) {
      d1[i] = v1->rp3[i] - v1->rp1[i];
      d2[i] = v2->rp3[i] - v2->rp1[i];
    }

    /* Find the cross products */
   dif = vcross(v1, ref, c1);
   dif = vcross(v2, ref, c2);

    /* Normalize the two vectors */
   m1 = m2 = 0.0;
   for (i = 0; i < 3; i++) {
      m1 += c1[i] * c1[i];
      m2 += c2[i] * c2[i];
    }

   m1 = (float)sqrt((double)m1);
   m2 = (float)sqrt((double)m2);

   for (i = 0; i < 3; i++) {
      c1[i] = c1[i]  / m1;
      c2[i] = c2[i]  / m2;
    }

    /* Compare the two vectors */
   dif = FALSE;
   for (i = 0; i < 3; i++)
      if (c1[i] != c2[i]) dif = TRUE;
```

```
    return(dif);
  } /* Routine is_cross */


void do_minr(proj, num, norm, h0, ref0, refm, thetamax, rot_plane, count)

struct tmp_rot proj[MAX_VERTS_FACE];                 /* Current face */
int     num;                                         /* Edges on the face */
float   norm[3];                                     /* Normal of the face */
float   h0;                                          /* height @ r=0 */
float   ref0[3],refm[3];                             /* Ref. clip vectors */
float   thetamax;                                    /* Clip theta */
struct rot_seg rot_plane[4*TOT_VERTS];               /* Clipped edges */
int     *count;                                      /* # clipped edges */

/* The routine do_minr computes the minimum radius of the shadow and
   performs the projections necessary to place the shadow into the
   rotational sweep plane. */

{
    float   refpt[3];                                /* Ref. pt. of minr line */
    float   step[3];                                 /* Para. step of line */
    float   loc[MAX_VERTS_FACE];                     /* Para. pts on line */
    int     code;                                    /* edge_clip rtn code */
    int     scode;                                   /* Rtn code of 1st edge */
    int     ptr;                                     /* Pointer of edge list */
    int     cnt;                                     /* Counts # intersect. */
    int     last;                                    /* Temporary memory */
    int     i,j;                                     /* Loop indices */
    float   swap;                                    /* Sorting variable */
    int     sptr;                                    /* Pointer to 1st edge */
    float   temp;                                    /* Temporary data */
    struct tmp_rot minr[MAX_VERTS_FACE];             /* Minr edges */
    float   tmax;                                    /* Actual thetamax */

    /* Find the parameters of the minimal radius line */
    code = getminline(proj[0].rp1, norm, refpt, step);

    /* Intersect the line with the face */
    if (code != 1) {
       /* Intersect with the first edge */
       ptr = cnt = 0;
       scode = code = edge_clip(&proj[ptr], refpt, step, &loc[cnt]);

       /* Skip over initial edges that are on the minimal radius line */
       while ((ptr < (num-1)) & (code == 4)) {
           ptr++;
           code = edge_clip(&proj[ptr], refpt, step, &loc[cnt]);
       }

       /* Remember this edge */
       sptr = ptr;

       /* Do the remaining edges */
       while (ptr < num) {

           /* Check the edge */
           code = edge_clip(&proj[ptr], refpt, step, &loc[cnt]);
```

```
        /* Determine the type of edge */
        if (code == 0) { cnt++;
                       }
        else if (code == 2) {
            /* Skip to edge of edge /vertex intersection pair */
            last = ptr;
            code = 4;
            while ((ptr < (num-1)) & (code == 4)) {
                ptr++;
                code = edge_clip(&proj[ptr], refpt, step, &loc[cnt]);
             }

            /* if not at end, process intersection */
            if (code != 4)
                if (is_cross(&proj[ptr], &proj[last], step) == TRUE) {
                    cnt++;
                  }
            else cnt = cnt - 1;

        } /* else if code = 2 */

         /* go to next edge */
        ptr++;
    } /* while ptr < num */

    /* check for processing over the closure */
    if ((code == 4) | (code == 2))
        if (is_cross(&proj[last], &proj[sptr], step) == TRUE) {
                cnt++;
              }
    else cnt = cnt - 1;

    /* Sort the parametric points */
    for (i = 0; i < (cnt - 1); i++)
        for (j = (i + 1); j < cnt; j++)
            if (loc[i] > loc[j]) {
                temp = loc[i];
                loc[i] = loc[j];
                loc[j] = temp;
            }

    /* Convert the parametric points to edges */
    ptr = 0;
    for (i = 0; i < cnt; i += 2) {
        /* Copy over the start, middle, and end points */
        temp = 0.5*(loc[i] + loc[i+1]);
        for (j = 0; j < 3; j++) {
            minr[ptr].rp1[j] = refpt[j] + step[j]*loc[i];
            minr[ptr].rp3[j] = refpt[j] + step[j]*loc[i+1];
            minr[ptr].rp2[j] = refpt[j] + step[j]*temp;
         }
        /* Compute the theta values */
        if ((minr[ptr].rp1[0] == 0.0) & (minr[ptr].rp1[1] == 0.0))
            minr[ptr].th[0] = 0.0;
        else
            minr[ptr].th[0] = (float)atan2((double)minr[ptr].rp1[1],
                                           (double)minr[ptr].rp1[0]);

        if ((minr[ptr].rp2[0] == 0.0) & (minr[ptr].rp2[1] == 0.0))
            minr[ptr].th[1] = 0.0;
        else
            minr[ptr].th[1] = (float)atan2((double)minr[ptr].rp2[1],
                                           (double)minr[ptr].rp2[0]);
```

```
            if ((minr[ptr].rp3[0] == 0.0) & (minr[ptr].rp3[1] == 0.0))
                minr[ptr].th[2] = 0.0;
            else
                minr[ptr].th[2] = (float)atan2((double)minr[ptr].rp3[1],
                                               (double)minr[ptr].rp3[0]);

            for (i = 0; i < 3; i++)
                if (minr[ptr].th[i] < 0.0)
                    minr[ptr].th[i] += 2*pi;
        } /* for i = 0 */

        /* clip the minimal radius edges */
        cnt  = cnt / 2;
        tmax = thetamax;

        if (thetamax > pi) {  /* Sweep in two sections */

            for (i = 0; i < cnt; i++)
                code = rot_clip(&minr[i], 1.0, ref0, ref0, pi, h0,
                                rot_plane, count);
            rot_reflect(minr, cnt);
            for (i = 0; i < cnt; i++)
                code = rot_clip(&minr[i], -1.0, ref0, ref0, pi, h0,
                                rot_plane, count);
            tmax = thetamax - pi;
        }

        for (i = 0; i < cnt; i++)
            code = rot_clip(&minr[i], 1.0, ref0, refm, tmax, h0,
                            rot_plane, count);

        rot_reflect(minr, cnt);
        for (i = 0; i < cnt; i++)
            code = rot_clip(&minr[i], -1.0, ref0, refm, tmax, h0,
                            rot_plane, count);

    } /* if code <> 1 */
} /* routine do_minr */
```

```
/*
**                          NOTICE OF COPYRIGHT
**              Copyright (C) Rensselaer Polytechnic Institute.
**                         1990 ALL RIGHTS RESERVED.
**
**
** Permission to use, distribute, and copy is granted ONLY
** for research purposes, provided that this notice is
** displayed and the author is acknowledged.
**
** This software is provided in the hope that it will be
** useful. BUT, in no event will the authors or Rensselaer
** be liable for any damages whatsoever, including any lost
** profits, lost monies, business interruption, or other
** special, incidental or consequential damages arising out
** of the use or inability to use (including but not
** limited to loss of data or data being rendered
** inaccurate or losses sustained by third parties or a
** failure of this software to operate) even if the user
** has been advised of the possibility of such damages, or
** for any claim by any other party.
**
** This software was developed at the facilities of the
** Center for Intelligent Robotic Systems for Space
** Exploration, Troy, New York, thanks to generous project
** funding by NASA.
**
** Package: Gripper Envelope Detection Using Sweep Shadows
**
** Written By: Henry L. Welch
**
*/
/***********************************************************************
 *  renvel.c  — This module contains the support routines to handle   *
 *                 the 2-D plane sweep for the gripper envelope problem. *
 *                 Rotational routines only.                            *
 *                                                                      *
 *  Written by: HLW November, 1989                                      *
 ***********************************************************************/
/* The following conventions are assumed throughout:

   1) Coordinates are seen as row matrices.
   2) Transformation matrices are right-multiplied.
*/

#include <stdio.h>
#include <math.h>
#include "envel.h"
#include "const.h"


void getrotref(rot, origin, thetam, pt, ref, ref0, refm)
float rot[3];                          /* rotation axis */
float origin[3];                       /* origin of rotation */
float thetam;                          /* theta maximum */
float pt[3];                           /* point to reference */
float ref[3][3];                       /* coord. sys. xform. */
float ref0[3];                         /* ref. vector for theta0 */
float refm[3];                         /* ref. vector for thetamax */

/* The routine getrotref determines the reference matrices for the
   theta=0 and theta=thetamax orientations of the rotational sweep
   plane. It requires an axis of rotation, the origin of the rotation,
   the extent of the rotation, and the reference point about
```

```
    which to base the rotational sweep. */
{
    int i;                                  /* loop index */
    float dot;                              /* dot product */
    float mag;                              /* vector magnitude */
    float mat[3][3];                        /* non-inverted coord. mtx */
    float diff[3];                          /* difference vector */
    float temp;                             /* temporary storage */
    float cm,sm;                            /* cosine and sine of thetamax */

    /* set-up the z-axis, compute the difference vector and ref dot product */
    mag = 0.0;
    for (i = 0; i < 3; i++)
        mag += rot[i]*rot[i];
    mag = (float)sqrt((double)mag);

    dot = 0.0;
    for (i = 0; i < 3; i++) {
        mat[2][i] = rot[i] /mag;
        diff[i] = pt[i] - origin[i];
        dot += diff[i]*mat[2][i];
    }

    /* compute the reference x-axis using a normalization procedure */
    mag = 0.0;
    for (i = 0; i < 3; i++) {
        mat[0][i] = diff[i] - dot*mat[2][i];
        mag += mat[0][i]*mat[0][i];
    }

    mag = (float)sqrt((double)mag);
    for (i = 0; i < 3; i++)
        mat[0][i] = mat[0][i] /mag;

    /* compute the reference y-axis by using a cross product */
    mat[1][0] = mat[2][1]*mat[0][2] - mat[2][2]*mat[0][1];
    mat[1][1] = mat[2][2]*mat[0][0] - mat[2][0]*mat[0][2];
    mat[1][2] = mat[2][0]*mat[0][1] - mat[2][1]*mat[0][0];

    /* compute the zero reference by inversion */
    mat3inv(mat, ref);

    /* compute the thetamax reference by rotation */
    if (thetam > pi) thetam = thetam - pi;
    cm = (float)cos((double)thetam);
    sm = (float)sin((double)thetam);

    /* assign the reference vectors for clipping */
    ref0[0] = 0.0;
    ref0[1] = 1.0;
    ref0[2] = 0.0;

    refm[0] = -sm;
    refm[1] = cm;
    refm[2] = 0.0;

} /* routine getrotref */

void get_norm (innorm, ref, outnorm)
```

```
struct coord3 innorm;                                /* original normal */
float   ref[3][3];                                   /* coord xform mtx */
float   outnorm[3];                                  /* xformed normal */

/* The routine get_norm transforms a planar normal vector from one
   rectangular coordinate system to another. */

{
    int i,j;                                         /* loop indices */

    /* perform the xform */
    for (i = 0; i < 3; i++) {
        outnorm[i] = 0.0;
        for (j = 0; j < 3; j++)
            outnorm[i] += innorm.pt[j]*ref[j][i];
    } /* for (i = */

} /* procedure get_norm */


void r0ht(proj, norm, h)
struct tmp_rot *proj;                                /* plane information */
float   norm[3];                                     /* plane normal */
float   *h;                                          /* height at r=0 */

/* The routine r0ht determines the height of a plane at the point
   where the cylindrical coordinate r = 0.

   NOTE: It is mathematically impossible for this routine to be called
         when norm[2] (ie nz) is 0.0 */

{
    int i;                                           /* loop index */

    /* compute the height using vector calculus */
    *h = 0.0;
    for (i = 0; i < 3; i++)
        *h =  *h + proj->rp1[i] * norm[i] /norm[2];

} /* procedure r0ht */


void r2p(pt, origin, ref, rp, th)

float pt[3];                                         /* original cart pt */
float origin[3];                                     /* local origin */
float ref[3][3];                                     /* coord xfrom mtx */
float rp[3];                                         /* xformed cart. pt */
float *th;                                           /* cylindrical theta */

/* The routine r2p converts a cartesian point from one reference frame
   to another and computes the cylindral theta of that point in the new
   reference frame. */

{
    int i,j;                                         /* loop indices */
    /* transform the point from one frame to the other */
    for (i = 0; i < 3; i++) {
        rp[i] = 0.0;
        for (j = 0; j < 3; j++)
            rp[i] += (pt[j] - origin[j])*ref[j][i];
    }
```

```
    /* compute the cylindrical theta */
    if ((rp[0] == 0.0) & (rp[1] == 0.0))
        *th = 0.0;
    else
        *th = (float)atan2((double)rp[1], (double)rp[0]);

    if (*th < 0.0) *th = 2*pi + *th;
} /* routine r2p */


void rot_prep (p1, p2, origin, ref, proj)

float  p1[3];                              /* starting endpoint */
float  p2[3];                              /* ending endpoint */
float  origin[3];                          /* local origin */
float  ref[3][3];                          /* coord. sys. xform */
struct tmp_rot *proj;                      /* projected data */

/* The routine rot_prep converts a line segment in one cartesian
   reference frame to another via a local origin and new basis vector
   set. The new cartesian coordinates are then projected to an angle
   in the cylindrical coordinate frame of the same orientation. */

{
    float tmp[3];                          /* temp midpoint */
    int i;                                 /* loop index */

    /* convert the endpoints */
    r2p(p1, origin, ref, proj->rp1, &proj->th[0]);
    r2p(p2, origin, ref, proj->rp3, &proj->th[2]);

    /* find and convert the line segments midpoint */
    for (i = 0; i < 3; i++)
        tmp[i] = 0.5 * (p1[i] + p2[i]);
    r2p(tmp, origin, ref, proj->rp2, &proj->th[1]);

} /* routine rot_prep */



void get_mpt(proj, ref, mpt)

struct tmp_rot *proj;                      /* temporary proj */
float  ref[3];                             /* reference vector */
float  mpt[3];                             /* clipped point */

/* The routine get_mpt clips the projected line segment described by
   proj to the angle defined by the vector ref. The resultant point
   is returned in mpt. */

{
    float dot1, dot2;                      /* dot products */
    int   i;                               /* loop index */
    float m;                               /* distance along seg. */

    /* use vector calculus to perform the clip */
    /* compute the two dot products */
    dot1 = dot2 = 0.0;
    for (i = 0; i < 3; i++) {
        dot1 += proj->rp1[i] * ref[i];
        dot2 += proj->rp3[i] * ref[i];
    }
```

```
    /* find the segment point on the clip angle */
    m = dot1 / (dot1 - dot2);

    for (i = 0; i < 3; i++)
      mpt[i] = proj->rp1[i] + m*(proj->rp3[i] - proj->rp1[i]);

  } /* routine get_mpt */


void rot_reflect(proj, count)

struct tmp_rot proj[MAX_VERTS_FACE];                    /* points to reflect */
int       count;                                        /* number of points */

{
    int i,j;                                            /* loop indices */

    /* reflect all thetas through the origin */
    for ( i = 0; i < count; i++)
        for (j = 0; j < 3; j++)
            if (proj[i].th[j] < pi)
                proj[i].th[j] += pi;
            else
                proj[i].th[j] += -pi;

  } /* routine rot_reflect */


void full_rot(proj, ref, rot_plane, count)

struct tmp_rot *proj;                                   /* segment to proj */
float   ref[3];                                         /* clip reference */
struct rot_seg rot_plane[4*TOT_VERTS];                  /* clipped points */
int       *count;                                       /* no of points */

 /* The routine full_rot performs the projection and clipping necessary
    for a rotation of exactly pi radians. */

{
    float mpt[3];                                       /* clipped point */
    int     i;                                          /* loop index */

    /* check for segment in first two quadrants only */
    if ((proj->th[0] <= pi) & (proj->th[2] <= pi)) {
        for (i = 0; i < 3; i++) {
            rot_plane[*count].pt[i]  = proj->rp1[i];
            rot_plane[*count].del[i] = proj->rp3[i] - proj->rp1[i];
          }
        rot_plane[*count].sign = 1.0;
        *count = *count + 1;
      }

    else /* check for segment in third and fourth quadrants only */
        if((proj->th[0] > pi) & (proj->th[2] > pi)) {
            for (i = 0; i < 3; i++) {
                rot_plane[*count].pt[i]  = proj->rp1[i];
                rot_plane[*count].del[i] = proj->rp3[i] - proj->rp1[i];
              }
            rot_plane[*count].sign = 1.0;
            *count = *count + 1;
          }
```


C-2

```
        else {  /* segment must be clipped at theta = 0,pi */
            get_mpt(proj, ref, mpt);

            /* save the clipped data */
            for (i = 0; i < 3; i++) {
                rot_plane[*count].pt[i]      = proj->rp1[i];
                rot_plane[*count].del[i]     = mpt[i] - proj->rp1[i];
                rot_plane[*count+1].pt[i]    = mpt[i];
                rot_plane[*count+1].del[i]   = proj->rp3[i] - mpt[i];
            }

            /* determine which half has the positive r values */
            if (proj->th[0] <= pi) {  /* first endpoint in quads 1 & 2 */
                rot_plane[*count].sign     = 1.0;
                rot_plane[*count+1].sign   = -1.0;
            }
            else {
                rot_plane[*count].sign     = -1.0;
                rot_plane[*count+1].sign   = 1.0;
            }

            *count = *count + 2;
        }  /* else seg must be clipped */

    }  /* routine full_rot */


int rot_clip(proj, sign, ref0, refm, thetamax, h0, rot_plane, count)

struct tmp_rot *proj;                                   /* segment to clip */
float   sign;                                           /* sign of r */
float   ref0[3];                                        /* 0 clip vector */
float   refm[3];                                        /* max clip vector */
float   thetamax;                                       /* maximum theta */
float   h0;                                             /* height @ r=0 */
struct rot_seg rot_plane[4*TOT_VERTS];                  /* clipped segments */
int     *count;                                         /* no of clipped segs */

/* The routine rot_clip performs the rotational clipping of an input
   segment.  Due to the complications caused by rotational wrap
   around and reflection through the origin, various return codes are
   used to denote the type of clip which took place. */

{
    float mpt[3];                                       /* clipped point */
    int   i;                                            /* loop index */
    int   rtn_code;                                     /* return code */

#define theta1 proj->th[0]
#define theta2 proj->th[1]
#define theta3 proj->th[2]

    /* save the sign */
    rot_plane[*count].sign = sign;

    /* Check for segment location relative to the theta wedge */

    if (theta1 <= thetamax) {
        if (theta3 <= thetamax) {

            /* Segment all inclusive and not clipped */
            for (i = 0; i < 3; i++) {
                rot_plane[*count].pt[i]   = proj->rp1[i];
                rot_plane[*count].del[i]  = proj->rp3[i] - proj->rp1[i];
```

```
                }
            *count = *count + 1;
            rtn_code = 1;
        }

        else {  /* clip theta3 */
            if ((theta2 > theta3) | (theta2 < theta1)) {
                /* clip theta3 to zero */
                get_mpt(proj, ref0, mpt);
                rtn_code = 16;
            }
            else {
                /* clip theta3 to thetamax */
                get_mpt(proj, refm, mpt);
                rtn_code = 24;
            }

            /* Store the clipped segment */
            for (i = 0; i < 3; i++) {
                rot_plane[*count].pt[i]  = proj->rp1[i];
                rot_plane[*count].del[i] = mpt[i] - proj->rp1[i];
            }
            *count = *count + 1;
        }  /* else clip theta3 */

    }  /* if theta1 <= thetamax */
    else {  /* clip theta1 */
        if (theta3 <= thetamax) {
            if ((theta2 > theta1) | (theta2 < theta3)) {
                /* clip theta1 to zero */
                get_mpt(proj, ref0, mpt);
                rtn_code = 4;
            }
            else {
                /* clip theta1 to thetamax */
                get_mpt(proj, refm, mpt);
                rtn_code = 6;
            }

            /* Save the clipped information */
            for (i = 0; i < 3; i++) {
                rot_plane[*count].pt[i]  = mpt[i];
                rot_plane[*count].del[i] = proj->rp3[i] - mpt[i];
            }
            *count = *count + 1;
        }  /* if theta3 > thetamax */

        else {  /* clip both theta1 and theta3 */
            if ((theta1 < theta3) & ((theta2 < theta1) | (theta2 > theta3))) {

                /* clip theta1 to thetamax, theta3 to zero */
                get_mpt(proj, refm, mpt);
                for (i = 0; i < 3; i++)
                    rot_plane[*count].pt[i] = mpt[i];

                get_mpt(proj, ref0, mpt);
                for (i = 0; i < 3; i++)
                    rot_plane[*count].del[i] = mpt[i] - rot_plane[*count].pt[i];

                *count = *count + 1;
                rtn_code = 22;
```

```
                } /* if theta1 < theta3 etc */

            else /* segment is not swept */
                rtn_code = 0;

            if ((theta3 < theta1) & ((theta2 < theta3) | (theta2 > theta1))) {

                /* clip theta1 to zero, theta3 to thetamax */
                get_mpt(proj, ref0, mpt);
                for (i = 0; i < 3; i++)
                    rot_plane[*count].pt[i] = mpt[i];

                get_mpt(proj, refm, mpt);
                for (i = 0; i < 3; i++)
                    rot_plane[*count].del[i] = mpt[i] - rot_plane[*count].pt[i];

                *count = *count + 1;
                rtn_code = 28;
            } /* if theta3 < theta1 etc */

            else {
                /* segment is not swept */
                rtn_code = 0;
            }
        } /* else clip both */

    } /* else clip theta1 */
    return(rtn_code);

} /* routine rot_clip */
```

```
/*
**                         NOTICE OF COPYRIGHT
**           Copyright (C) Rensselaer Polytechnic Institute.
**                      1990 ALL RIGHTS RESERVED.
**
**
** Permission to use, distribute, and copy is granted ONLY
** for research purposes, provided that this notice is
** displayed and the author is acknowledged.
**
** This software is provided in the hope that it will be
** useful. BUT, in no event will the authors or Rensselaer
** be liable for any damages whatsoever, including any lost
** profits, lost monies, business interruption, or other
** special, incidental or consequential damages arising out
** of the use or inability to use (including but not
** limited to loss of data or data being rendered
** inaccurate or losses sustained by third parties or a
** failure of this software to operate) even if the user
** has been advised of the possibility of such damages, or
** for any claim by any other party.
**
** This software was developed at the facilities of the
** Center for Intelligent Robotic Systems for Space
** Exploration, Troy, New York, thanks to generous project
** funding by NASA.
**
** Package: Gripper Envelope Detection Using Sweep Shadows
**
** Written By: Henry L. Welch
**
*/
/*************************************************************************
 *                                                                       *
 * rotate.c -- This module contains the higher level routines which     *
 *                  perform the 2-D plane sweep.  Most of the support   *
 *                  routines used are contained in the module envel.c   *
 *                                                                       *
 * Written by: Henry L. Welch November, 1989                            *
 *                                                                       *
 *************************************************************************/

#include <stdio.h>
#include <math.h>
#include "envel.h"
#include "const.h"
#include "data.h"

void rsweep(proj, num, h0, ref0, refm, thetamax, sign, rot_plane, count)

struct tmp_rot proj[MAX_VERTS_FACE];          /* rect. proj. of face */
int     num;                                  /* number of verts */
float   h0;                                   /* heigt @ r=0 */
float   ref0[3];                              /* theta 0 ref. vector */
float   refm[3];                              /* thetamax ref vector */
float   thetamax;                             /* max theta of rotation */
float   sign;                                 /* sign of the radii */
struct rot_seg rot_plane[4*TOT_VERTS];        /* clipped segments */
int     *count;                               /* number of segments */

/* The routine rsweep performs the sweeping of a planar face through a
   rotational wedge in cylindrical coordinate system. */

{
```

```
    int     start;                                          /* flag for no segs yet   */
    int     first;                                          /* Pointer into rot_plane */
    int     ccode;                                          /* Current clip code */
    int     scode;                                          /* First seg. clip code */
    int     pcode;                                          /* Previous clip code */
    int     incase;                                         /* Temporary pointer */
    int     i,j;                                            /* Loop indices */
    float   last[3];                                        /* Last included point */

    /* initialize pointers and flags, then loop on all edges */
    start = TRUE;
    first = *count;
    for (i = 0; i < num; i++) {
        /* clip the edge */
        incase = *count;
        ccode = rot_clip(&proj[i], sign, ref0, refm, thetamax, h0, rot_plane,
                         count);

        /* determine the type of clip */
        if (ccode != 0) {  /* edge has a segment */

            if (start == TRUE) {  /* Process first edge with a segment */
                start = FALSE;
                /* Save relevant data of clip code */
                if ((ccode == 1) | (ccode == 16) | (ccode == 24))
                    scode = 0;
                else if ((ccode == 4) | (ccode == 28))
                    scode = 1;
                else scode = 2;
            }  /* if start = True */
            else {
                /* check for type of clipping */
                if ((ccode == 4) | (ccode == 28)) {  /* Entering at theta = 0 */
                    if ((pcode == 16) | (pcode == 22)) {  /* left at theta = 0 */

                        /* close along theta = 0 */
                        for (j = 0; j < 3; j++) {
                            rot_plane[*count].pt[j]  = last[j];
                            rot_plane[*count].del[j] = rot_plane[*count - 1].pt[j] -
                                                       last[j];
                        }
                        rot_plane[*count].sign = sign;
                        *count = *count + 1;
                    }
                    else {  /* left at theta = thetamax */

                        /* close via r = 0 */
                        for (j = 0; j < 2; j++) {
                            rot_plane[*count].pt[j]  = last[j];
                            rot_plane[*count].del[j] = -last[j];

                            rot_plane[*count+1].pt[j]  = 0.0;
                            rot_plane[*count+1].del[j] = rot_plane[*count-1].pt[j];
                        }

                        rot_plane[*count].pt[2]  = last[2];
                        rot_plane[*count].del[2] = h0 - last[2];

                        rot_plane[*count+1].pt[2]  = h0;
                        rot_plane[*count+1].del[2] = rot_plane[*count-1].pt[2] - h0;

                        rot_plane[*count].sign = rot_plane[*count+1].sign = sign;
                        *count = *count + 2;
```

```
                } /* else left at theta = thetamax */

            } /* if coming in at theta = 0 */
            else if ((ccode == 6) | (ccode == 22)) { /* Enter @ thetamax */
                if ((pcode ==16) | (pcode == 22)) { /* Left at theta = 0 */

                    /* close via r = 0 */
                    for (j = 0; j < 2; j++) {
                        rot_plane[*count].pt[j]   = last[j];
                        rot_plane[*count].del[j] = -last[j];

                        rot_plane[*count+1].pt[j]   = 0.0;
                        rot_plane[*count+1].del[j] = rot_plane[*count-1].pt[j];
                    }

                    rot_plane[*count].pt[2]   = last[2];
                    rot_plane[*count].del[2] = h0 - last[2];

                    rot_plane[*count+1].pt[2]   = h0;
                    rot_plane[*count+1].del[2] = rot_plane[*count-1].pt[2] - h0;

                    rot_plane[*count].sign = rot_plane[*count+1].sign = sign;
                    *count = *count + 2;
                } /* if left at theta = 0 */
                else { /* left at thetamax */

                    /* close along thetamax */
                    for (j = 0; j < 3; j++) {
                        rot_plane[*count].pt[j]   = last[j];
                        rot_plane[*count].del[j] = rot_plane[*count - 1].pt[j] -
                                                   last[j];
                    }
                    rot_plane[*count].sign = sign;
                    *count = *count + 1;
                } /* else left at thetamax */
            } /* if entered at thetamax */
        } /* else check for type of clip */
        /* Process clip on exit if it exists */
        if ((ccode == 16) | (ccode == 24) | (ccode == 22) |
            (ccode == 28)) {
            pcode = ccode;
            for (j = 0; j < 3; j++) /* save last valid point */
                last[j] = rot_plane[incase].pt[j] +
                          rot_plane[incase].del[j];
        } /* if clip on exit */

    } /* if ccode <> 0 */
} /* for i = 0 */

if (start == FALSE) { /* close clip from beginning */

    if (((scode == 1) & ((pcode == 16) | (pcode == 22))) |
        ((scode == 2) & ((pcode == 24) | (pcode == 28)))) {

        /* clip on one side only */
        for (j = 0; j < 3; j++) {
            rot_plane[*count].pt[j]   = last[j];
            rot_plane[*count].del[j] = rot_plane[first].pt[j] -
                                       last[j];
        }
        rot_plane[*count].sign = sign;
        *count = *count + 1;
```

```
            } /* if clip on one side */

            else if (((scode == 1) & ((pcode == 24) | (pcode == 28))) |
                    ((scode == 2) & ((pcode == 16) | (pcode == 22)))) {

                /* close via r = 0 */
                for (j = 0; j < 2; j++) {
                    rot_plane[*count].pt[j]   = last[j];
                    rot_plane[*count].del[j]  = -last[j];

                    rot_plane[*count+1].pt[j]  = 0.0;
                    rot_plane[*count+1].del[j] = rot_plane[first].pt[j];
                }
                rot_plane[*count].pt[2]   = last[2];
                rot_plane[*count].del[2]  = h0 - last[2];

                rot_plane[*count+1].pt[2]  = h0;
                rot_plane[*count+1].del[2] = rot_plane[first].pt[2] - h0;

                rot_plane[*count].sign = rot_plane[*count+1].sign = sign;
                *count = *count + 2;
            } /* close via r = 0 */

        } /* if start = false */
} /* routine rsweep */


void do_rsweep (verts, num_objects, f_count, v_count, norms,
                origin, ref, ref0, refm, thetamax, rot_plane, count)

struct coord3 verts[MAX_OBJECTS][MAX_FACES][MAX_VERTS_FACE];  /* obj verts */
int     num_objects;                                    /* number of objects */
int     f_count[MAX_OBJECTS];                           /* face count */
int     v_count[MAX_OBJECTS][MAX_FACES];                /* vert count per face */
struct coord3 norms[MAX_OBJECTS][MAX_FACES];            /* normals for faces */
float   origin[3];                                      /* rotation origin */
float   ref[3][3];                                      /* inverted basis */
float   ref0[3];                                        /* theta 0 ref vector */
float   refm[3];                                        /* thetamax ref vector */
float   thetamax;                                       /* maximum theta */
struct rot_seg rot_plane[4*TOT_VERTS];                  /* rotational sweep */
int     *count;                                         /* segments in sweep */

/* The routine do_rsweep performs a rotational sweep on the environment. */

{
    int     i,j,k,l;                                    /* loop indices */
    int     vc;                                         /* vertex count */
    float   p1[3],p2[3];                                /* segment endpoints */
    float   tmax;                                       /* actual thetamax */
    float   norm[3];                                    /* face normal */
    float   h0;                                         /* height @ r=0 */
    struct tmp_rot proj[MAX_VERTS_FACE];                /* temp. data */

    /* initialize the count and loop on each object */
    *count = 0;
    for (i = 0; i < num_objects; i++) {

        /* loop on each face */
        for (j = 0; j < f_count[i]; j++) {
            vc = v_count[i][j];
```

```
            /* loop on each edge /vertex */
            for (k = 0; k < (vc - 1); k++) {

                /* copy over edge endpoints */
                for (l = 0; l < 3; l++) {
                    p1[l] = verts[i][j][k].pt[l];
                    p2[l] = verts[i][j][k+1].pt[l];
                }

                /* prepare the edge */
                rot_prep(p1, p2, origin, ref, &proj[k]);
            } /* for k = 0 */
            /* close the face */
            for (l = 0; l < 3; l++) {
                p1[l] = verts[i][j][vc - 1].pt[l];
                p2[l] = verts[i][j][0].pt[l];
            }

            rot_prep(p1, p2, origin, ref, &proj[vc-1]);

            /* sweep the plane rotationally */
            get_norm(norms[i][j], ref, norm);
            r0ht(&proj[0], norm, &h0);

            /* process the minimal radius shadow line */
            do_minr(proj, vc, norm, h0, ref0, refm, thetamax, rot_plane, count);

            /* Sweep the face */
            tmax = thetamax;

            if (thetamax > pi) { /* Must sweep in two sections */
                rsweep(proj, vc, h0, ref0, ref0, pi, 1.0, rot_plane, count);
                rot_reflect(proj, vc);
                rsweep(proj, vc, h0, ref0, ref0, pi, -1.0, rot_plane, count);
                tmax = thetamax - pi;
            }
            rsweep(proj, vc, h0, ref0, refm, tmax, 1.0, rot_plane, count);
            rot_reflect(proj, vc);
            rsweep(proj, vc, h0, ref0, refm, tmax, -1.0, rot_plane, count);

        } /* for j = 0 */
    } /* for i = 0 */
} /* do_rsweep */


void analyze_rot_plane(rot_plane, count, site, distance)

struct rot_seg rot_plane[4*TOT_VERTS];          /* rotational sweep results */
int     count;                                  /* number of segments */
float   site[2];                                /* grasp site coords */
float   *distance;                              /* distance to nearest point */

/* The routine analyze_rot_plane finds the nearest linear distance from the
   grasp site to the sweep segments. It is done by approximating the (r,h)
   curves by ten straight line segments. */

{
    int     i,j,k;                              /* loop indices */
    float   pt[3];                              /* current segment point */
    float   del[3];                             /* segment coord steps */
```

```
    float   r;                                    /* radius of point */
    float   temp[2];                              /* temporary storage */
    float   dist;                                 /* intermediate distance */
    struct seg line;                              /* convenient data structure */

    /* initialize the distance to a ridiculous value and loop on all segments */
    *distance = 1000000000.0;

    for (i = 0; i < count; i++) {

        /* copy over the segment information */
        for (j = 0; j < 3; j++) {
            pt[j]  = rot_plane[i].pt[j];
            del[j] = 0.1 * rot_plane[i].del[j];
        }

        /* Break the curve into 10 straight line segments */
        line.p2[0] = (float)sqrt((double)(pt[0]*pt[0] + pt[1]*pt[1])) *
                         rot_plane[i].sign;
        line.p2[1] = pt[2];

        for (j = 0; j < 10; j++) {

            /* set-up the beginning point */
            line.p1[0] = line.p2[0];
            line.p1[1] = line.p2[1];

            /* move ahead the delta */
            for (k = 0; k < 3; k++)
                pt[k] += del[k];

            /* set-up the ending point */
            line.p2[0] = (float)sqrt((double)(pt[0]*pt[0] + pt[1]*pt[1])) *
                             rot_plane[i].sign;
            line.p2[1] = pt[2];

            /* check the distance and update accordingly */
            d_pt2seg(site, line, &dist, temp);

            if (dist < *distance) *distance = dist;
        } /* for j = */
    } /* for i = */

    *distance = (float)sqrt((double)*distance);

} /* analyze_rot_plane */
```

```
/*
**                      NOTICE OF COPYRIGHT
**          Copyright (C) Rensselaer Polytechnic Institute.
**                    1990 ALL RIGHTS RESERVED.
**
**
** Permission to use, distribute, and copy is granted ONLY
** for research purposes, provided that this notice is
** displayed and the author is acknowledged.
**
** This software is provided in the hope that it will be
** useful. BUT, in no event will the authors or Rensselaer
** be liable for any damages whatsoever, including any lost
** profits, lost monies, business interruption, or other
** special, incidental or consequential damages arising out
** of the use or inability to use (including but not
** limited to loss of data or data being rendered
** inaccurate or losses sustained by third parties or a
** failure of this software to operate) even if the user
** has been advised of the possibility of such damages, or
** for any claim by any other party.
**
** This software was developed at the facilities of the
** Center for Intelligent Robotic Systems for Space
** Exploration, Troy, New York, thanks to generous project
** funding by NASA.
**
** Package: Gripper Envelope Detection Using Sweep Shadows
**
** Written By: Henry L. Welch
**
*/
/**********************************************************************
 *                                                                    *
 * sweep.c — This module contains the higher level routines which    *
 *               perform the 2-D plane sweep.  Most of the support    *
 *               routines used are contained in the module envel.c    *
 *                                                                    *
 * Written by: Henry L. Welch June&July, 1989                         *
 *                                                                    *
 **********************************************************************/

#include <stdio.h>
#include <math.h>
#include "envel.h"
#include "const.h"
#include "data.h"

void do_tsweep (origin, basis, length,
                num_objects, f_count, v_count, verts,
                plane, count)
float origin[3];                             /* Defn of traj. sweep */
float basis[3][3];                           /* Inverted basis */
float length;

int    num_objects;                          /* Defn of environment */
int    f_count[MAX_OBJECTS];                 /* Face count /object */
int    v_count[MAX_OBJECTS][MAX_FACES];      /* Vertex count /face */
struct coord3 verts[MAX_OBJECTS][MAX_FACES]
                   [MAX_VERTS_FACE];             /* Extracted vertices */
```

```
struct seg plane[2*TOT_VERTS];                          /* Swept plane */
int      *count;                                          /* Number of segments */

/* The routine do_tsweep performs a trajectory sweep of the trajectory
   specified by origin, basis, and length.  The objects swept are those
   defined using num_objects ... verts.  The resulting swept segments
   are contained in the structure plane. */

{
   int     i,j,k,l;                                      /* Loop indices */
   int     start;                                         /* New face flag */
   int     first;                                         /* First segment on face */
   int     swept;                                          /* Condition flag */
   int     temp;                                           /* Temporary index */
   float p1[3],p2[3];                                    /* Segment endpoints */

   /* start at the beginning and loop on all objects */
   *count = 0;
   for (i = 0; i < num_objects; i++) {
      /* loop for each face */
      for (j = 0; j < f_count[i]; j++) {
         start = 1;
         first = *count;

         /* loop for each vertex pair */
         for (k = 0; k < (v_count[i][j] - 1); k++) {
            /* Copy over edge endpoints */
            for (l = 0; l < 3; l++) {
               p1[l] = verts[i][j][k].pt[l];
               p2[l] = verts[i][j][k+1].pt[l];
            }

            /* Sweep the edge */
            swept = sweepseg(origin, basis, length, p1, p2, &plane[*count]);
            if (swept == 1) {                         /* The edge was swept */
               if (start == 1) start = 0;      /* Face is in plane */

               else {                                 /* Check for continuity */
                  if ((plane[*count-1].p2[0] != plane[*count].p1[0]) |
                        (plane[*count-1].p2[1] != plane[*count].p1[1])) {

                     /* Connect the segments over the clipped vertex */
                     *count = *count + 1;
                     plane[*count].p1[0] = plane[*count-2].p2[0];
                     plane[*count].p1[1] = plane[*count-2].p2[1];
                     plane[*count].p2[0] = plane[*count-1].p1[0];
                     plane[*count].p2[1] = plane[*count-1].p1[1];
                  } /* if ((plane */
               } /* else */
               *count = *count + 1;                  /* count the added segment */
            } /* if (swept */
         } /* for (k */

         if (start == 0) {                            /* try closing edge */

            /* copy over first and last vertices */
            for (l = 0; l < 3; l++) {
               p1[l] = verts[i][j][v_count[i][j]-1].pt[l];
               p2[l] = verts[i][j][0].pt[l];
            }
```

```
                /* Sweep the edge */
                swept = sweepseg(origin, basis, length, p1, p2, &plane[*count]);

                if (swept == 1) {                         /* The edge was swept */
                    temp = *count;                         /* This may be needed */
                    if ((plane[*count-1].p2[0] != plane[*count].p1[0]) ||
                         (plane[*count-1].p2[1] != plane[*count].p1[1])) {

                        /* Connect the segments over the clipped vertex */
                        *count = *count + 1;
                        plane[*count].p1[0] = plane[*count-2].p2[0];
                        plane[*count].p1[1] = plane[*count-2].p2[1];
                        plane[*count].p2[0] = plane[*count-1].p1[0];
                        plane[*count].p2[1] = plane[*count-1].p1[1];
                    } /* if ((plane */

                    /* Check the other end */
                    if ((plane[first].p1[0] != plane[temp].p2[0]) ||
                         (plane[first].p1[1] != plane[temp].p2[1])) {

                        /* Connect the segments over the clipped vertex */
                        *count = *count + 1;
                        plane[*count].p1[0] = plane[temp].p2[0];
                        plane[*count].p1[1] = plane[temp].p2[1];
                        plane[*count].p2[0] = plane[first].p1[0];
                        plane[*count].p2[1] = plane[first].p1[1];
                    } /* if ((plane */
                    *count = *count + 1;                     /* count the added segment */
                } /* if (swept */
            } /* if (start */
        } /* for (j */
    } /* for (i */

} /* routine do_tsweep */

void extract_verts (num_objects, objs, object, xfm_verts,
                    f_count, v_count, verts)
int     num_objects;                                    /* number of objects */
int     objs[MAX_OBJECTS];                               /* Objects in use */
struct n_object object[MAX_OBJECTS];                   /* Object data */
struct coord3 xfm_verts[MAX_OBJECTS][MAX_VERTS];       /* Transformed verts */
int     f_count[MAX_OBJECTS];                            /* Face count /object */
int     v_count[MAX_OBJECTS][MAX_FACES];                /* Vertex count /face */
struct coord3 verts[MAX_OBJECTS][MAX_FACES][MAX_VERTS_FACE];

/* The routine extract_verts aligns the transformed object vertices
   into a more useful format. */

{
    int     i,j,k,l;                                     /* Loop indices */
    int     current_obj;                                 /* Object in use */

    /* Loop on the objects */
    for (i = 0; i < num_objects; i++) {
        current_obj = objs[i];

        /* Loop on all faces */
        f_count[i] = object[current_obj].num_faces;
        for (j = 0; j < f_count[i]; j++) {

            /* Loop on all vertices */
            v_count[i][j] = object[current_obj].vert_count[j];
            for (k = 0; k < v_count[i][j]; k++)
```

```
                /* copy over the vertex coordinates */
                for (l = 0; l < 3; l++)
                    verts[i][j][k].pt[l] = xfm_verts[current_obj]
                                          [object[current_obj].vert_list[j][k]].pt[l];
        } /* for (j */
    } /* for (i */

} /* routine extract_verts */

void xfm_objects(num_objects, objs, object, xfm_verts, xfm_norms)
int     num_objects;                                  /* Number of objects */
int     objs[MAX_OBJECTS];                            /* Objects in use */
struct n_object object[MAX_OBJECTS];                  /* The objects */
struct coord3 xfm_verts[MAX_OBJECTS][MAX_VERTS];      /* Transformed vertices */
struct coord3 xfm_norms[MAX_OBJECTS][MAX_FACES];      /* Transformed normals */

/* The routine xfm_objects scales, rotates, and translates the objects
   in the environment to the appropriate location in the assembly */

{
    int     i,j,k;                                    /* Loop indices */
    int     current_obj;                              /* Object being xformed */
    float sc_mat[4][4];                               /* Scaling matrix */
    float ph_mat[4][4];                               /* Phi rotation matrix */
    float th_mat[4][4];                               /* Theta rot. matrix */
    float tr_mat[4][4];                               /* Translation matrix */
    float t_mat1[4][4], t_mat2[4][4], xfm_mat[4][4];  /* Combined matrices */
    struct t_coord4 onorm;                            /* Temp storage */

    /* Loop on the objects */
    for (i = 0; i < num_objects; i++) {
        current_obj = objs[i];

        /* Prepare transformation matrices */
        for (j = 0; j < 4; j++) {
            for (k = 0; k < 4; k++) {
                sc_mat[j][k] = 0.0;
                ph_mat[j][k] = 0.0;
                th_mat[j][k] = 0.0;
                tr_mat[j][k] = 0.0;
            }
            sc_mat[j][j] = 1.0;
            ph_mat[j][j] = 1.0;
            th_mat[j][j] = 1.0;
            tr_mat[j][j] = 1.0;
        } /* for (j */

        for (j = 0; j < 3; j++) {
            sc_mat[j][j] = object[current_obj].scale;
            tr_mat[3][j] = object[current_obj].xlate[j];
        }

        th_mat[1][1] = (float)cos((double)object[current_obj].theta);
        th_mat[1][2] = (float)sin((double)object[current_obj].theta);
        th_mat[2][1] = -th_mat[1][2];
        th_mat[2][2] = th_mat[1][1];

        ph_mat[0][0] = (float)cos((double)object[current_obj].phi);
        ph_mat[0][1] = (float)sin((double)object[current_obj].phi);
        ph_mat[1][1] = ph_mat[0][0];
        ph_mat[1][0] = -ph_mat[0][1];
```

```
        /* Combine the matrices */
        mat4mul(sc_mat, ph_mat, t_mat1);
        mat4mul(th_mat, tr_mat, t_mat2);
        mat4mul(t_mat1, t_mat2, xfm_mat);

        /* Transform the vertices for this object */
        for (j = 0; j < object[current_obj].num_verts; j++)
            do_xfm (xfm_mat, object[current_obj].vertices[j],
                    &xfm_verts[i][j]);

        /* Transfrom the face normals for this object */
        mat4mul(ph_mat, th_mat, xfm_mat);
        for (j = 0; j < object[current_obj].num_faces; j++) {

            /* Prepare the appropriate data type */
            onorm.x = object[current_obj].normals[j].x;
            onorm.y = object[current_obj].normals[j].y;
            onorm.z = object[current_obj].normals[j].z;
            onorm.w = 1.0;

            /* Transform the normal */
            do_xfm(xfm_mat, onorm, &xfm_norms[i][j]);
        }

    } /* for (i */
} /* routine xfm_objects */


void analyze_plane (plane, count, type, site, normal, distance)
struct seg plane [2*TOT_VERTS];            /* sweep plane */
int     count;                             /* segments in the plane */
int     type;                              /* type of area to consider */
float   site[2];                           /* location of grasp site */
float   normal[2];                         /* half-plane normal */
float   *distance;                         /* heuristic distance returned */

/* The routine analyze_plane, scans a sweep plane of segments to determine the
   square of the distance to the nearest obstruction. Whether the entire
   plane or only half is considered depends on the flag type. */

{   struct seg oplane[2*TOT_VERTS];        /* halved plane */
    int     ocount;                        /* number of segments */
    float   dist;                          /* temporary distance */
    int     i;                             /* a loop index */
    float   pt[2];                         /* nearest point on segment */

    /* determine if only half the plane need be considered */
    if (type == 1)   /* halve the plane */
        half_plane(plane, count, site, normal, oplane, ocount);
    else
        ocount = count;


    /* find the distance to the first point */
    if (type == 1)
        d_pt2seg(site, oplane[0], &dist, pt);
    else
        d_pt2seg(site, plane[0], &dist, pt);

    *distance = dist;
```

```
    /* process the remaining points */
    for (i = 1; i < ocount; i++) {
        if (type == 1)
            d_pt2seg(site, oplane[i], &dist, pt);
        else
            d_pt2seg(site, plane[i], &dist, pt);

        if (dist < *distance) *distance = dist;
    }

    /* take the square root of the result */
    *distance = (float)sqrt((double)*distance);

} /* routine analyze_plane */
/* *(splane + 3) */
```

```
/*
**                        NOTICE OF COPYRIGHT
**            Copyright (C) Rensselaer Polytechnic Institute.
**                   1990 ALL RIGHTS RESERVED.
**
**
** Permission to use, distribute, and copy is granted ONLY
** for research purposes, provided that this notice is
** displayed and the author is acknowledged.
**
** This software is provided in the hope that it will be
** useful. BUT, in no event will the authors or Rensselaer
** be liable for any damages whatsoever, including any lost
** profits, lost monies, business interruption, or other
** special, incidental or consequential damages arising out
** of the use or inability to use (including but not
** limited to loss of data or data being rendered
** inaccurate or losses sustained by third parties or a
** failure of this software to operate) even if the user
** has been advised of the possibility of such damages, or
** for any claim by any other party.
**
** This software was developed at the facilities of the
** Center for Intelligent Robotic Systems for Space
** Exploration, Troy, New York, thanks to generous project
** funding by NASA.
**
** Package: Gripper Envelope Detection Using Sweep Shadows
**
** Written By: Henry L. Welch
**
*/
/*********************************************************************
 * sweepio.c  — This module contains the input/output support       *
 *                routines for the 2-D plane sweep used in the       *
 *                gripper envelope problem.                          *
 *                                                                   *
 * Written by:    HLW August, 1989                                   *
 *********************************************************************/
#include <stdio.h>
#include <math.h>
#include "envel.h"
#include "const.h"
#include "data.h"

void printmat(mat)
float mat[3][3];                               /* The matrix to print */

 /* The routine printmat prints out a 3x3 matrix to the terminal screen */

{
    int i,j;                                   /* Loop indices */

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++)
            printf ("%f       ",mat[i][j]);
        printf ("\n");
        }
  } /* End routine printmat */

void dump_plane(plane, count, traj)
```

```
struct seg plane[2*TOT_VERTS];                  /* Plane to print */
int count;                                      /* Number of segments */
int traj;                                       /* Trajectory number */

/* The routine dump_plane sends a graph /sunplot compatible version
   of the swept plane data to the file "dump"+traj */

{
    int   i;                                    /* Loop index */
    FILE *f;                                    /* File pointer */
    char *num[6];                               /* file number */

    /* identify the trajectory number */
    sprintf(num, "dump%d", traj);
    f = fopen(num, "w");                        /* Open the file */

    for (i = 0; i < count; i ++)
        fprintf(f, "%f %f %f %f \" \" \n", plane[i].p1[0], plane[i].p1[1],
                                            plane[i].p2[0], plane[i].p2[1]);

    fclose(f);

 } /* routine dump_plane */


void read_plane(plane, count)
struct seg plane[2*TOT_VERTS];                  /* Plane to print */
int *count;                                     /* Number of segments */

/* The routine dump_plane sends a graph /sunplot compatible version
   of the swept plane data to the file "dump" */

{
    char   i[50];                               /* Junk string */
    FILE *f;                                    /* File pointer */

    f = fopen("dump", "r");                     /* Open the file */

    *count = 0;
    while (!feof(f)) {
        fscanf(f, "%f%f%f%f%s%s", &plane[*count].p1[0], &plane[*count].p1[1],
                                 &plane[*count].p2[0], &plane[*count].p2[1],
                                 i, i);
        *count = *count + 1;
     }

    fclose(f);

 } /* routine read_plane */

void read_CAD (num_objects, object, objs)

int       *num_objects;                         /* number of objects */
struct    n_object object[MAX_OBJECTS];         /* objects */
int       objs[MAX_OBJECTS];                    /* objects in use */

/* The routine read_CAD inputs the CAD data to be used by the sweep
   algorithm. */

{
    char     infil[80];                         /* CAD file name */
```

```
    FILE    *inptr;                                    /* file pointer */
    int     i,j,k;                                     /* loop indices */
    float   *pn;                                       /* a useful pointer */

    /* read in object */
    inptr = 0;
    while (inptr == 0) {
        printf("Enter the name of the CAD data file: ");
        scanf("%s", infil);
        if ((inptr = fopen(infil, "r")) == 0)
            printf("error opening input file\n");
    }

    /* read number of objects */
    fscanf(inptr, "%d", num_objects);

    /* for each object */
    for (i = 0; i < *num_objects; i++) {

        /* read number of polygons and faces */
        fscanf(inptr, "%d %d", &object[i].num_verts, &object[i].num_faces);

        /* read in the theta, phi, and scale factors for the object */
        fscanf(inptr, "%f %f %f", &object[i].theta, &object[i].phi,
                                  &object[i].scale);

        /* read in the translational position of the object */
        fscanf(inptr, "%f %f %f", &object[i].xlate[0], &object[i].xlate[1],
                                  &object[i].xlate[2]);

        /* read in the coordinates for each face */
        for (j=0; j<object[i].num_verts; j++) {
            fscanf(inptr, "%f %f %f", &object[i].vertices[j].x,
                   &object[i].vertices[j].y, &object[i].vertices[j].z);
            object[i].vertices[j].w = 1.0;
        }

        /* read in the vertex list and normals for each face */
        for (j=0; j<object[i].num_faces; j++) {
            fscanf(inptr, "%d", &object[i].vert_count[j]);
            for (k=0; k<object[i].vert_count[j]; k++)
                fscanf(inptr, "%d", &object[i].vert_list[j][k]);
            fscanf(inptr, "%f %f %f", &object[i].normals[j].x,
                   &object[i].normals[j].y, &object[i].normals[j].z);
        }
        /* Make all objects in use */
        objs[i] = i;

    } /* for (i */
    fclose(infil);
} /* routine read_CAD */


void dump_rot_plane(rot_plane, count, traj)

struct rot_seg rot_plane[4*TOT_VERTS];          /* rotational sweep result */
int     count;                                  /* number of segments */
int     traj;                                   /* trajectory number */

/* The routine dump_rot_plane creates a sunplot readable file for displaying
   the results of the current rotational sweep. */

{
```

```
    int     i,j,k;                                    /* loop indices */
    FILE    *f;                                       /* file pointer */
    char    *num[6];                                  /* file name */
    float pt[3];                                      /* current point */
    float del[3];                                     /* segment step values */
    float r;                                          /* computed radius */

    /* identify and open the output file */
    sprintf(num, "dump%d", traj);

    f = fopen(num, "w");

    /* for each rotational segment */
    for (i = 0; i < count; i++) {

        /* extract the segment data */
        for (j = 0; j < 3; j++) {
            pt[j]   = rot_plane[i].pt[j];
            del[j]  = 0.1 * rot_plane[i].del[j];
        }

        /* break the curve into 10 straight line segments */
        for (j = 0; j < 10; j++) {
            r = (float)sqrt((double)(pt[0]*pt[0] + pt[1]*pt[1])) *
                    rot_plane[i].sign;
            fprintf(f, "%f %f \n", r, pt[2]);

            /* update the point information */
            for (k = 0; k < 3; k++)
                pt[k] += del[k];
        } /* for j */

        /* print out the last segment */
        r = (float)sqrt((double)(pt[0]*pt[0] + pt[1]*pt[1])) * rot_plane[i].sign;
        fprintf(f, "%f %f \" \" \n", r, pt[2]);
    } /* for i */

    fclose(f);
} /* routine dump_rot_plane */
```

```
/*****************************************************************************
 *                                                                           *
 * test.c -- This is the sample call frame for the plane sweep software.     *
 *                                                                           *
 * Written by: HLW June 1990                                                 *
 *                                                                           *
 *****************************************************************************/

#include <stdio.h>
#include "envel.h"
#include "const.h"
#include "data.h"
```

main

```
main()
{
float   coord[3][3];            /* sweep coord system */
float   inv[3][3];              /* inverted coord system */
float   origin[3];              /* sweep origin */
float   axis[3];                /* rotation axis */
float   refpt[3];               /* reference point */
float   ref0[3];                /* theta 0 reference vector */
float   refm[3];                /* theta max reference vector */
float   len;                    /* length of the sweep */
float   distance;               /* heuristic distance in plane */
float   site[2];                /* location of grasp in plane */
float   norm[2];                /* grasp site normal */
int     ttype;                  /* trajectory type */
int     type;                   /* grasp type */
```

```
int      num_grasps;                                   /* number of grasp sites */
int      i,j,k;                                        /* loop indices */
FILE     *inptr;                                       /* input file pointer */
char     infil[80];                                    /* input filename */
struct   n_object object[MAX_OBJECTS];                 /* objects to process */
int      num_objects;                                  /* number of objects */
int      objs[MAX_OBJECTS];                            /* object usage list */
int      f_count[MAX_OBJECTS];                         /* face count per object */
int      v_count[MAX_OBJECTS][MAX_FACES];              /* vertex count per face */
struct   coord3 verts[MAX_OBJECTS][MAX_FACES][MAX_VERTS_FACE];
struct   seg plane[2*TOT_VERTS];                       /* swept plane */
struct   rot_seg rot_plane[4*TOT_VERTS];               /* rotated plane */
int      count;                                        /* segments in swept plane */
struct   coord3 xfm_verts[MAX_OBJECTS][MAX_VERTS];
struct   coord3 xfm_norms[MAX_OBJECTS][MAX_FACES];
int      num_traj;                                     /* number of trajectories */

    /* input the objects */
    read_CAD (&num_objects, object, objs);

    /* open the trajectory file */
    inptr = 0;
    while (inptr == 0) {
        printf("Enter the name of the file containing sweep info: ");
        scanf("%s", infil);
        if ((inptr = fopen(infil, "r")) == 0)
            printf("error opening file!\n");
    }

    /* read in the objects to be used in this scan */
    fscanf(inptr, "%d", &num_objects);

    for (i = 0; i < num_objects; i++)
        fscanf(inptr, "%d", &objs[i]);

    /* preprocess the objects so they are where they belong */
    xfm_objects(num_objects, objs, object, xfm_verts, xfm_norms);

    /* extract the vertices */
    extract_verts(num_objects, objs, object, xfm_verts, f_count,
                  v_count, verts);

    /* Find the trajectory sweep information. */
    /* read the number of trajectories */
    fscanf (inptr, "%d", &num_traj);

    /* for each trajectory */
    for (i = 0; i < num_traj; i++) {

        /* read in the type of sweep 0=line 1=rotate */
        fscanf(inptr, "%d", &ttype);

        if (ttype == 0) {

            /* read in the reference coordinate system for the trajectory */
            for (j = 0; j < 3; j++)
                fscanf(inptr, "%f", &origin[j]);

            for (j = 0; j < 3; j++)
                for (k = 0; k < 3; k++)
                    fscanf (inptr, "%f", &coord[j][k]);
```

```
            /* read in the length of the trajectory */
            fscanf(inptr, "%f", &len);

            /* invert the coordinate system */
            mat3inv(coord, inv);

            /* perform the sweep */
            do_tsweep(origin, inv, len, num_objects, f_count, v_count, verts,
                        plane, &count);

            /* save the sweep data */
            dump_plane(plane, count, i);
        } /* if (ttype == 0 */

        else { /* process a rotation */

            /* read in the origin of the rotation */
            for (j = 0; j < 3; j++)
                fscanf(inptr, "%f", &origin[j]);

            /* read in the rotation axis */
            for (j = 0; j < 3; j++)
                fscanf(inptr, "%f", &axis[j]);

            /* read in the reference point */
            for (j = 0; j < 3; j++)
                fscanf(inptr, "%f", &refpt[j]);

            /* read in the angle of rotation in radians */
            fscanf(inptr, "%f", &len);

            /* set up the various reference vectors etc */
            getrotref(axis, origin, len, refpt, inv, ref0, refm);

            /* handle the rotational sweep */
            do_rsweep(verts, num_objects, f_count, v_count, xfm_norms,
                        origin, inv, ref0, refm, len, rot_plane, &count);
printf(" TOTAL: %d\n",count);
            /* save the data for plotting */
            dump_rot_plane(rot_plane, count, i);
        } /* else */

        /* evaluate each grasp site for suitability */
        fscanf(inptr, "%d", &num_grasps);

        for (j =0; j < num_grasps; j++) {

            /* Read in each grasp site and process it */
            fscanf(inptr, "%d %f %f", &type, &site[0], &site[1]);

            if (type == 1)    /* read in the normal */
                fscanf(inptr, "%f %f", &norm[0], &norm[1]);

            if (ttype == 0)
                analyze_plane(plane, count, type, site, norm, &distance);
            else
                analyze_rot_plane(rot_plane, count, site, &distance);

            printf (" The distance for segment %d, grasp %d is: %f \n",
                    i, j, distance);
```

```
            } /* end for (j = 0 */

        } /* end for (i = 0 */

    fclose(infil);
} /* end main */
```